UNIVERSITY OF CALIFORNIA, SAN DIEGO

Democratic Community-based Search with XML Full-Text Queries

A dissertation submitted in partial satisfaction of the
requirements for the degree
Doctor of Philosophy

in

Computer Science

by

Emiran Curtmola

Committee in charge:

Professor Alin Deutsch, Chair
Professor Richard K. Belew
Professor James D. Hollan
Professor Yannis Papakonstantinou
Professor Victor Vianu
Research Scientist Sihem Amer-Yahia

2009

The dissertation of Emiran Curtmola is approved, and it is acceptable in quality and form for publication on microfilm and electronically:

_____

_____

_____

_____

_____

_____

Chair

University of California, San Diego

2009

DEDICATION

to my parents, my brother, and all my friends

EPIGRAPH

Time is a great teacher, but unfortunately it kills all its pupils.

—Louis Hector Berlioz

TABLE OF CONTENTS

References . . . 184

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGEMENTS

I would like to thank everybody I have worked with during my graduate studies that have helped me understand and learn the principles of doing good research in general, and bring my work to a successful completion. They are all special people that have opened my eyes and mind, that have challenged me and then let me challenge them back contributing to fruitful discussions, and that have encouraged me throughout my Ph.D. studies.

I would like to acknowledge and especially thank Alin Deutsch, my thesis advisor, for his invaluable guidance and continuous support over the past six years. He is one of the smartest people I know. I did my best to follow in his footsteps and learn everything from him. I hope I could be as enthusiastic, passionate, and energetic as Alin. I would like to acknowledge my committee members too for their support and insightful comments including Sihem Amer-Yahia, Richard Belew, James Hollan, Yannis Papakonstantinou, and Victor Vianu.

I am very grateful to have met the following wonderful people with whom I worked while I spent three summers as a research intern at AT&T Labs Research, especially Sihem Amer-Yahia, Philip Brown, Mary Fernandez, K.K. Ramakrishnan, and Divesh Srivastava. Particularly, I want to thank Sihem who has been my inspiration. She has helped me explore new things and has pointed me to fresh topics that turned out to be paramount research directions.

Also, I would like to thank Dionysions Logothetis and Kenneth Yocum for their assistance with the infrastructure support to build a distributed platform for the XTreeNet demonstration.

I would like to thank my collaborators and co-authors from IBM Research Almaden: Andrey Balmin, Latha Colby, Fatma Őzcan, Quanzhong Li, Sharath Srinivas, and Zografoula Vagena. This has been again an excellent place for great research and for great people where I spent two summers as a research intern.

Last but not least, I would like to thank my lab mates in the Database group at University of California, San Diego who always showed great support and proper amount of attention required to stimulate interesting discussions. I was very lucky to be part of this lively environment. I also thank all my colleagues in the department of Computer Science as well as my paper reviewers from various conferences.

Chapter 2 is currently being prepared for submission for publication of the material, which is joint work with Alin Deutsch, K.K. Ramakrishnan, and Divesh Srivastava. The dissertation author was the primary investigator and author of the paper.

Chapter 3, in part, is a revised reprint of the following materials published in the SIGMOD XIME-P 2005 Workshop and WWW 2005 Special interest tracks and posters, which are joint works with Sihem Amer-Yahia, Philip Brown, and Mary Fernandez [61, 62]. The dissertation author was the primary investigator and author of the papers.

Chapter 4, in part, is a revised reprint of the following material published in SIGMOD 2006 Conference, which is joint work with Sihem Amer-Yahia and Alin Deutsch [25]. This material is also currently being prepared for submission for publication. The dissertation author was the primary investigator and author of the papers.

Chapter 5, in part, is a revised reprint of the material published in VLDB 2008 Conference, which is joint work with Alin Deutsch, Dionysios Logothetis, K.K. Ramakrishnan, Divesh Srivastava, and Kenneth Yocum [64]. The material was accompanied by a real implementation and a system demonstration during the Demonstration track at VLDB 2008 Conference.

VITA

2003	Bachelor of Science in Computer Science and Engineering, Polytechnic University of Bucharest, Romania
2006	Master of Science in Computer Science, University of California, San Diego
2009	Doctor of Philosophy in Computer Science, University of California, San Diego

PUBLICATIONS

Andrey Balmin and Emiran Curtmola, "WikiAnalytics: Ad-hoc Querying of Highly Heterogeneous Structured Data". In *International Conference on Data Engineering* (ICDE'10) Demonstration, March 2010.

Andrey Balmin, Latha Colby, Emiran Curtmola, Quanzhong Li, and Fatma Őzcan. "Search Driven Analysis of Heterogeneous XML Data". In *Conference on Innovative Data Systems Research* (CIDR'09), January 2009.

Emiran Curtmola, Alin Deutsch, Dionysios Logothetis, K.K. Ramakrishnan, Divesh Srivastava, and Kenneth Yocum. "XTreeNet: Democratic Community Search". In *International Conference on Very Large Data Bases* (VLDB'08) Demonstration, August 2008.

Andrey Balmin, Latha Colby, Emiran Curtmola, Quanzhong Li, Fatma Őzcan, Sharath Srinivas, and Zografoula Vagena. "SEDA: A System for Search, Exploration, Discovery and Analysis of XML Data". In *International Conference on Very Large Data Bases* (VLDB'08) Demonstration, August 2008.

Emiran Curtmola. "A Platform for Search in the Big Web 2.0". In *SIGMOD 2007 PhD Workshop on Innovative Database Research* (SIGMOD IDAR'07), June 2007.

Sihem Amer-Yahia, Emiran Curtmola, and Alin Deutsch. "Flexible and Efficient XML Search with Complex Full-Text Predicates". In *ACM SIGMOD International Conference on Management of Data* (SIGMOD'06), June 2006.

Nicola Onose, Alin Deutsch, Yannis Papakonstantinou, and Emiran Curtmola. "Rewriting Nested XML Queries using Nested Views". In *ACM SIGMOD International Conference on Management of Data* (SIGMOD'06), June 2006.

Emiran Curtmola, Sihem Amer-Yahia, Philip Brown, and Mary F. Fernández. "GalaTex: A Conformant Implementation of the XQuery Full-Text Language". In *International Workshop on XQuery Implementation, Experience and Perspectives* (SIGMOD XIME-P'05), June 2005.

Emiran Curtmola, Sihem Amer-Yahia, Philip Brown, and Mary F. Fernández. "GalaTex: A Conformant Implementation of the XQuery Full-Text Language". In *International World Wide Web Conference - Special interest tracks and posters* (WWW'05), May 2005.

Emiran Curtmola, Sihem Amer-Yahia, and Alin Deutsch. "Implementation and Open Research Issues in XML Full-Text Search". In *New York Area DB/IR Day Workshop*, April 2005.

FIELDS OF STUDY

Database Systems

ABSTRACT OF THE DISSERTATION

Democratic Community-based Search with XML Full-Text Queries

by

Emiran Curtmola
Doctor of Philosophy in Computer Science
University of California San Diego, 2009
Professor Alin Deutsch, Chair

As the web evolves, it is becoming easier to form online communities based on shared interests, and to create and publish data on a wide variety of topics. With this democratization of information creation, it is natural to query, in an ad-hoc and expressive fashion, the global collection that is the union of all local data collections of others within the community. In order to publish and locate documents of interest while fully delivering on the promise of free data exchange, any community-supporting infrastructure needs to enforce the key requirement to preserve privacy of the association of content providers with potential sensitive information. This privacy-preserving publishing requirement prevents censorship, harassment, or discrimination of users by third parties. It also precludes some obvious approaches that reuse and build on existing centralized technologies including search engines and hosted online communities.

This dissertation facilitates democratization of data publishing and efficient search with powerful full-text queries over the community global collection by means of a novel distributed framework that disseminate queries in online communities. We address two challenging issues that arise in this context: the design of distributed access methods to publishers and the evaluation of expressive queries (i.e., XML full-text) locally at the publisher thereof.

First, given the virtual nature of the global data collection, we study the problem of efficiently discovering publishers in the community that contain documents matching a user query. We call such peers relevant publishers. We propose a

novel distributed infrastructure in which data resides only with the publishers owning it. The infrastructure disseminates user queries to publishers, who answer them at their own discretion, under data-location anonymity constraints. That is the query forwarding infrastructure prevents leaking information about which publishers are capable of answering a certain query.

Second, once queries reach relevant publishers, we study how they efficiently process the incoming queries over their local repositories. Given that the commonly used data model for information exchange on the Web is semi-structured (e.g., XML), we propose algorithms for the evaluation and optimization of expressive XML queries that integrate structured and full-text search, including the W3C XQuery Full-Text standard.

Chapter 1

Introduction

1.1 Democratic Publishing with Expressive Search

The recent explosion of various types of information being generated from so many different sources under a large variety of social interactions between users has made search a hot topic for many research communities. We are witnessing a revolutionary process of democratization of information creation on the web in the sense that it is easier to create and publish data on a wide variety of topics; this is evident from the proliferation of blogs (e.g., Blogger [2], WordPress [21], LiveJournal [13]), wikis (e.g., Wikipedia [20]), user-generated content, social sites, etc. In a way, publishing and querying for content in Web-based communities has become a day-to-day convenience. Moreover, it is easier to have publishers organize in ad-hoc communities based on shared interests, which we call communities of interest (rather than of coincidence). This is true if we consider the popularity and rapid growth of social networking sites like Facebook [4], Twitter [19], MySpace [14], LinkedIn [12], and Friendster [5].

With the confluence of these trends comes the natural desire of users to freely exchange data within the community - this includes making one's own data accessible to others within the community, and also be able to query, tag, and comment on the rich global community collection that is the union of all local data collections of users in the community. In principle, a publishing system contains two components. The publishing component allows publishers to make their data accessible for querying

within the community; while the querying component allows users to query the global collection.

However, the data published in online communities is in principle distributed among the participating users depending on where it was created or was shared, and depending on who owns it. Another area which faces an analogous problem is that of digital library management where high-function interoperability is a must in order to facilitate access and search across remote online library catalogs. Moreover, this data reside in various representations. Similarly, many applications of scientific and collaborative nature, including projects like GEON [6] and BIRN [1], depend on data sets that reside in different locations, that are managed with different systems, and that are accessed via different query languages. For a long time, the ability of such applications and of their users to exchange data and issue ad-hoc queries (i.e., queries on the actual content of the collection) over communities of distributed collections of data sources was hindered by the lack of a homogeneous data model at the sources. The popularity of XML [111] as the *de facto* standard format for electronic data exchange on the Web, has determined the majority of data sources to adopt and export their data as XML regardless of how they actually store it. For instance, all major commercial relational database management systems provide full support to export an XML query interface now. Consequently, XML has emerged as the natural candidate for the homogeneous data model, fuelling yet again the research in the database integration area. In the rest of the thesis, we assume without any loss of generality that individual users publish, store, and export collections of documents, usually in XML format. By documents we understand any of the following: articles, news, tables, blogs, tags, comments, tweets, or anything of rich textual content.

At the same time, XML is a natural way to represent on the Web what is called semi-structured data. For years, databases and information retrieval have evolved independently as two fields with different agendas: databases manage structured data or records, while information systems manage unstructured or text documents. However, the explosion of data in the recent years has laid down the way for an increasing number of applications that require querying data based on both the structure and the textual content of documents. As hinted above, most of the enterprise content and data on the Web, including user generated content, digital

libraries, and scientific databases, are neither strictly relational nor purely textual – but semi-structured. In addition, whereas traditional Web search focused on simple keyword search, nowadays more online applications require complex features posing a critical need for expressive, relevant, and flexible search tools.

Therefore, in an effort to provide better retrieval performance (both in terms of efficiency and effectiveness) on large collections, we face a new requirement: the desirability to search the community data with powerful queries involving composable full-text search predicates. We call these queries XML full-text queries. To meet this requirement, a recent W3C effort extended XQuery, the recommended language for querying XML based only on the document structure, to querying XML data on both text and structure. This effort produced the XQuery Full-Text [114] standard, also known as XQFT, which allows to issue complex and expressive queries that go beyond simple keyword search by using full-text capabilities. These capabilities include a rich set of predicates that are textual conditions expressed in terms of keyword positions in the document, ranging from Boolean search to combining sophisticated proximity distance and keyword-order conditions on keywords, and on semi-structured content in general. For instance, a legitimate full-text query on the Wikipedia data might ask for

> (‡) *all documents that contain terms related to* `official ethnic` *groups and* `minority` *languages that occur within a window of 10 words, with* `ethnic` *appearing before* `minority` [120, 114].

Moreover, recent events have shown that it is necessary to consider a new feature that is critical to nowadays publishing in online communities – that of allowing the users to freely exchange data and therefore, of enabling freedom of speech online without the fear of retribution. Nowadays, for many of us, the Internet is all about free search for information. People have come to learn that their online blogs along with the mainstream news websites can be easily censored, or worse, the true identity behind their online nicknames can be revealed. This is frightening if we think that this information can be used to censor or discriminate certain individuals pertaining to various online activist groups or dissidents, including common online communities whose scope is to merely raise awareness, to disagree, or simply to address sensitive issues related to politics, race, religion, gender identity, ethnicity, acts of charity, etc.

Therefore, in order to fully deliver on the promise of freely exchanging data, any community-supporting infrastructure needs to enforce the key requirement to preserve privacy of publishers' association with personal intention and with potential sensitive or controversial information. That is, there should be no easy way for any third party to infer or to collude published data to reveal the identity of all publishers containing documents on certain topics. Since this information can be further used to deny users access to certain publishers and their content or to deny selected publishers access online, the above requirement plays a paramount role in enabling freedom of speech online and therefore, in protecting users from censorship, harassment, and possible discrimination by third parties, be they of governmental, corporate, or other nature (including advertising, marketing, and tracking companies).

This privacy-preserving publishing requirement precludes some obvious approaches that reuse and build on existing conventional centralized technologies, e.g., search engines, hosted online communities, etc. The centralized approach assumes that the community data can be crawled, harvested, and collected at one single site which is responsible for handling all user queries. At the same time, users presume these technologies to have the acceptable duty to protect their expectation of anonymity or privacy, which is not necessarly true in the presence of network attacks, subpoena, or other court orders. More, whereas this online content delivery and social networks phenomenon is growing, it has sparked debates on subjects including net neutrality, content ownership and control, etc. While these are designed to handle a large number of potential publishers and the dynamic nature of published data, enabling a straightforward query access to the global data collection, the downside is that publishers are disintermediated from consumers by the central site. That is that publishers loose their autonomy as they need to relinquish their data. Moreover, the central site has control over the visibility of publishers to user queries, and the publishers lose control over who is accessing and who is interested in their content. For instance, Facebook stores each and every photo uploaded, indefinitely, even if the user removes it or deletes the entire account from the website; same with the user profile information that may include real name, age, sex, friends list, etc. It is scary to consider how much content control there is in the hands of proprietary companies.

In addition, centralized solutions may be unsatisfactory if we consider aspects such as the crawling technology they use. The limitation relies in the quantity of data these crawlers can harvest on the Web. A popular trend is to query online data sources that are hidden behind query forms that we call the deep web. However, in practice, a crawler engine can index only the surface web, which is the data visible online, while it ignores entirely the deep web, which is of several orders of magnitude larger. More, these technologies suffer from insufficient data timeliness (i.e., the freshness of query results depends on the crawling frequency) when timely information dissemination to applications is a key feature. As well, centralized technologies are prone to traffic bottlenecks and represent single points of failure; therefore, congestion and server downtime could affect data availability among community users [1]. Relatedly, at the current rate of information creation and sharing online, centralized systems will not be able to scale indefinitely and are easy to attack.

At the same time, the emergence of the GRID and Peer-to-peer (P2P) technologies, can make the network itself act as an enabler to expose the vast amount of existing information which is virtual (i.e., not materialized at one location) and overcome these limitations. Nowadays, the popularity and diversity of such applications for content sharing and distribution including web caching (e.g., Akamai), file sharing (e.g., Gnutella, BitTorrent), multimedia sharing (e.g., Youtube), and VOIP (e.g., Skype) indicate many benefits to fully distributed P2P systems. One particular advantage of these systems is that any two users can communicate without the need of a centralized server site. This kind of communication makes it harder to completely block and does not require a single server to stay in service for the duration of a possible attack conflict over user or data privacy. Another advantage is that these systems make up a platform that cannot be owned. As such, we employ P2P technology as a viable alternative with lower cost for real-time community data search and resource discovery. Given the need for a flexible network, and given the need to accommodate new requirements for emerging applications, the Internet represents indeed an attractive infrastructure to support community-based democratic (i.e., free data exchange) data publishing among the world-wide users.

[1] http://www.techcrunch.com/2009/08/06/serious-twitter-outage-ongoing/, http://www.techcrunch.com/2009/09/01/gmail-now-really-down-can-i-get-my-email-back-please/

Consequently, due to the proliferation of online communities and social networking sites, the increasing number of applications in need for powerful search (e.g., beyond simple keyword search) and the need for democratic information exchange, there is a growing demand for democratization of data publishing in online communities. To this end, we study techniques to efficiently search expressively over distributed XML data sources and locate documents of interest while guaranteeing free speech online among the community users. We illustrate our problem statement in Example 1.1.1 below. The main focus of this dissertation is to design and implement a novel distributed framework, which empowers community users to join democratic communities, to publish information, and to search ad-hoc the community data using powerful XML full-text queries.

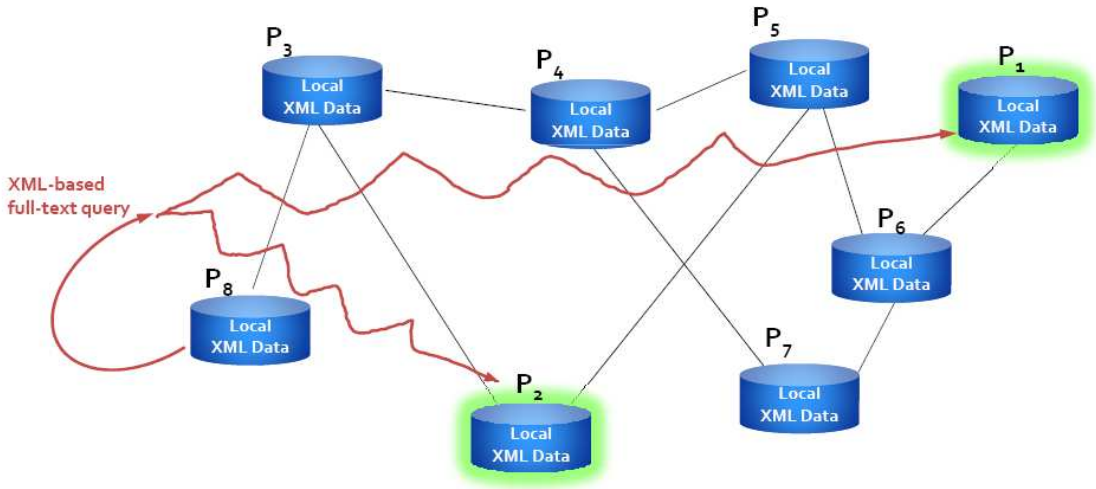


Figure 1.1: Example of a publishing system: Consumer P_8 poses an XML-based full-text query Q against the community data; the network identifies the relevant publishers P_1 and P_2 that can potentially answer Q; publishers P_1, P_2 run locally an evaluation engine for Q and return matching documents to P_8.

EXAMPLE 1.1.1. *To illustrate our novel distributed framework, we introduce a running example. Figure 1.1 contains the high-level description of a democratic publishing system for a virtual newspaper and blogging community. We consider a network of eight peer nodes, P_1 to P_8, connected in a mesh topology. Each peer publishes and stores a collection of XML documents (these are potentially virtual*

documents, physically stored in different formats). We call this to be an XML peer network. The global community data is virtual in the sense that the entire data published in the community is not actually materialized at any central site, but data resides with the publishing node only. In fact, the local data stored at a peer can be accessed via an XML text search interface by outside applications, i.e., a peer runs a local evaluation engine capable of answering expressive full-text queries over its XML store. These queries are posed against the community data using one of the XML full-text languages. In particular, we use the W3C XQuery Full-Text standard query language. The result of a user query is a set of matching documents which can originate from one or multiple sources and which satisfy all the query conditions. We call a peer that initiates queries over the community data to be a consumer or querier. Similarly, we call a peer providing content in the community to be a content provider or publisher. Each peer can be both a publisher and a consumer, and it can answer queries over its own local XML store at its own discretion, on behalf of queries initiated by other peers. For instance, in the figure, peer node P_8 poses a query to the community while the network identifies all relevant publisher nodes, P_1 and P_2, respectively, that can potentially store matching documents. When the query reaches P_1 or P_2, then the publisher runs a query evaluation engine locally over its local store and returns any matching documents to the querier. Since the publishing system is decentralized, each node has only a local view of the global community data, which is virtual. As such, it is hard to identify and pinpoint all publishers that can answer to a certain user query without extensively colluding with or attacking community members. Therefore, intuitively, this architecture is an enabler for online freedom of speech and for democratically searching the community data. ◇

We argue that successful search in democratic online communities faces several challenges that we outline in the next section.

1.2 Challenges

Given the virtual nature of the global data collection, it is challenging to provide democratic online community-based search with expressive querying, on both fronts: the design of the underlying back-end infrastructure which allows publishing

in the community as well as data source discovery, and the search platform over the document collection locally at the source. Given these goals, there are a few interesting challenges we need to meet.

One of the main challenges in querying distributed data sources is that of source discovery nature and scalability. In such a setting, data sources are peers who join and leave the community autonomously and keep their data locally without copying it to a central store. In general, there is a large number of decentralized publishers and an even larger number of consumers. Enabling such a publishing infrastructure involves addressing the following two key questions for the users: "whom to tell?", and "whom to ask?." First, publishers leaving or joining online communities would want to know whom to announce their presence and whom to tell about the availability of their data among the host of potential consumers. Second, consumers would want to know whom to ask for information and whom to send their queries to among the myriad of publishers. In this setting, it is critical to have the network locate the publishers that advertise content matching the user queries in a data-location anonymous way.

At the same time, any publishing system faces the performance challenge of making the data accessible and available for search in the community in an efficient manner. Given the size of the peer network, a naive flooding-based evaluation strategy that sends the query Q to all peers is impractical, resulting in the unnecessary evaluation of Q at peers whose local data is irrelevant to Q. On the performance front, the efficient querying of such distributed peers requires spreading the computation across the network by exploiting its structure to avoid congestion at the nodes.

Yet another challenge is of information discovery and processing nature, that of providing expressive query access to searching over individual data sources assuming that each source comprises of an XML database store. We have witnessed recently a lack of comprehensive solutions for search on semi-structured data. This process has been pushing the databases (DB) and the information retrieval (IR) communities toward a unified methodology. At one extreme, there is structured search which is mostly relational (i.e., limited to attribute value search) or is just based on XML trees. At the other extreme, there is IR-style fuzzy search on unstructured

data. It is therefore necessary to address the problem of designing new expressive ways to access semi-structured information by combining state-of-the-art DB XML retrieval and IR techniques that will enable to query XML data both on text and structure [63]. Thus, given the plethora of available publishers, where peers use non-uniform descriptions, with variable structure and textual content, and given the need to search over XML databases expressively, we employ XML queries with full-text predicates.

Recently, we have assisted at an explosion of competing proposals for XML full-text search languages, each with different expressive power, with different semantics, and often with fuzzy scoring [41]. We focus on an important class of these languages that we call the XQFT-class, after the most expressive of them, the W3C XQuery Full-Text standard, including languages and search systems like NEXI [120], XIRQL [75], JuruXML [46], XSearch [55], XRank [80], XKSearch [125], Schema-Free XQuery [92], etc. While these proposals come with efficient query evaluation, they are limited to only conjunctive keyword search (i.e. no predicates) over XML databases, or to full-text predicates in isolation. In this context, we distinguish the XQuery Full-Text (XQFT) to be a comprehensive extension of the XQuery language that accounts for full-text predicates allowing complex search based on both the structure and the textual part of documents.

Given that there is no comprehensive solution that combines state-of-the-art keyword search with full-text predicates over XML databases, it is critical to address their efficient evaluation on both tag structure and text contents, together with effective relevance ranking. There is a need for a universal optimization framework, in which to reason uniformly about the XQFT-class of languages and therefore, to guarantee the universality of on optimization solution.

Finally, it is non-trivial to ensure true free information exchange in online communities and, in particular, in nowadays publishing systems. In general, such systems require a variety of mechanisms to achieve democratic exchange of information that include ensuring user anonymity, encrypting communication channels, and data-location anonymization techniques. Whereas the first two are well studied in systems like Tor [68] and FreeNet [54], they are dependant on the encryption scheme strength and weaknesses, and require dedicated trusted servers to encrypt and route

the traffic through established anonymous tunnels. In practice, it is non-trivial to maintain and to protect these servers dynamically. Therefore, our infrastructure is necessary and it complements their resistance with another mechanism that preserves privacy of users in publishing communities in the sense that it provides anonymity for the association of users with certain information. We call this ability as *data-location anonymity* or DLA and we talk more about it in Chapter 2. In essence, our requirement insures that the network dissemination infrastructure does not leak information about which are the publishers capable of answering a certain class of queries. It is thus challenging to reason about data-location anonymity and how this affects the system design in order to minimize information known at a peer node in case of network breaches at nodes.

To summarize, the main challenges in democratic search in online communities with complex queries are: (1) locating relevant publishers that advertise data matching with a specified query and returning the corresponding matching documents while (2) ensuring the privacy of publishers' association with their advertised information against third parties, and (3) local evaluation, at the publisher, of queries over semi-structured data that combine structure and text search with complex and composable full-text predicates.

1.3 Solution

In this section, we address the challenges mentioned above and we give an overview of our methodology. In the following example, we illustrate a high-level view of our distributed access method solution as shown in Figure 1.2.

At high-level, we propose a distributed infrastructure in which data resides only with the publishers who own it. Our solution is based on allowing the publishers of the community to publish, store, and keep complete control over their own data; thus, enabling user autonomy in the community. Let us remind the community setting from Example 1.1.1 which contains eight publishers. Users can pose ad-hoc complex XML full-text queries (e.g., XQFT) into the network over the entire contents of the community data. That is, the queries are virtually posed against the local contents of all participating publishers of the community. For instance, in

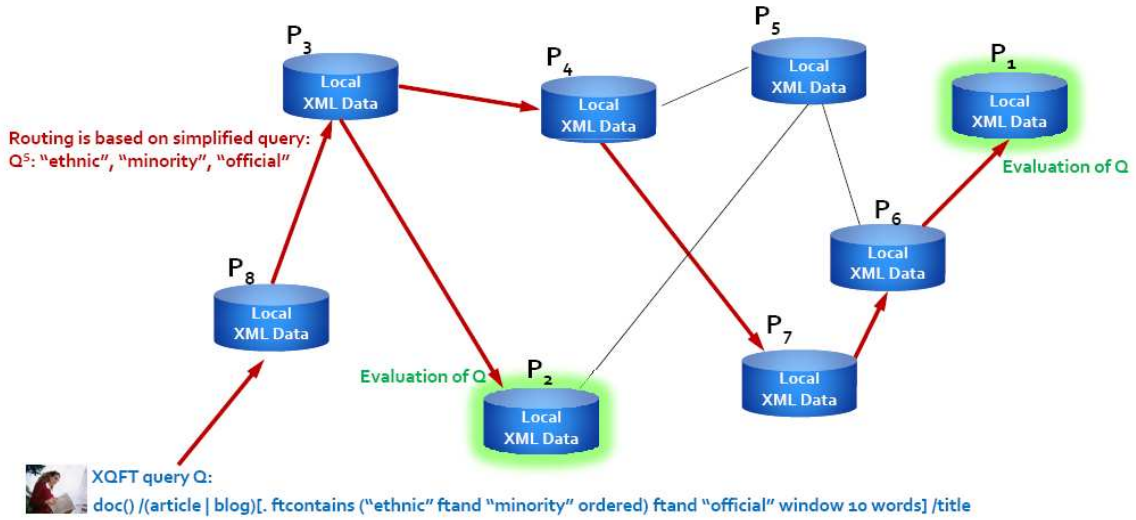


Figure 1.2: Distributed access methods: (i) query dissemination in the community network and (ii) full query evaluation at source

Figure 1.2, consumer P_8 poses query Q (as defined in ‡ page 3), this time expressed in the XQFT language:

$$Q : doc()/(article \mid blog)[.\ \textbf{ftcontains}\ (\text{“}ethnic\text{”}\ \textbf{and}\ \text{“}minority\text{”}\ \textbf{ordered})\ \textbf{and}\ \text{“}official\text{”}\ \textbf{window} \leq 10\ words]/title$$

We adopt a query dissemination approach where the user queries are forwarded in the network from consumers to publishers, who answer them with matching documents at their own discretion. Therefore, query answering has two principal parts. In the first step, the system routes query Q in the network and discovers only the data sources of interest, based on a simplified form of the query Q^S (i.e., the conjunction of query keywords from Q) and on a smart data index at the peer nodes. In our example, routing of Q in the network reaches the relevant publishers P_1 and P_2 by hopping through various community nodes. The second step handles how to query the local data repository for all publishers reached during the query routing process. These publishers decide based on their own access policy whether and how to release the matching documents back to the consumers. In order to find matching documents with Q, P_1 and P_2 evaluate the full query Q on their corresponding local document collections using an XQFT query processor.

We detail more on these two steps for answering queries in democratic communities in the following two sections.

1.3.1 Query Routing in the Community

We describe now our design principles and solution for distributed access methods to discover relevant publishers and thus, to enable efficient querying the global community data. For more details, please see Chapter 2. When a user query Q is posted in the community, our goal is to quickly identify which are the relevant sources to access that contain documents matching with Q. To this end, we design a novel distributed data index which assists the online community to discover data sources. Our index directs user queries only to relevant sources so that only those sites are accessed that can contribute to the query answer; therefore, minimizing unnecessary generated network traffic and computation overhead at nodes. The index can be maintained with little overhead as peers join or leave the network, or as their local XML store changes.

In order to efficiently disseminate queries, the index is a tree-shaped logical overlay over the peers in the network, which we call a query dissemination tree (QDT). That is, user peers are logically organized as nodes in hierarchical tree structures. A user query initiated by a peer is sent to the root of the dissemination tree, whence it is propagated down the tree. For example, in Figure 1.2, query Q is sent to root node at P_8. The arrow links show the dissemination of Q and identify the QDT used to route Q down the tree to the relevant publishers, P_1 and P_2, respectively. At every node, Q is evaluated on the peer's local store as explained in Section 1.3.2, then the peer checks whether the query should be forwarded to the peers who are its children in this QDT. For this purpose, each peer node in the tree maintains a small summary of the data advertised in the collection of documents in its subtrees. This summary serves to detect when no peer in the subtree can satisfy the query, whose forwarding can be aborted, thus efficient pruning the entire subtree.

Moreover, to address the performance challenges, our index infrastructure is organized as a union of overlay (i.e., logical) networks, called UQDT – a union of QDT's – as shown in Chapter 2. We show how to use the UQDT infrastructure

to maximize the overall throughput by balancing the congestion of queries observed at the peers given a workload of rich-expressive queries. While different queries might hit the same set of nodes, our goal is to balance the generated community search traffic (i.e., the number of messages) at nodes during query dissemination while preserving both low index space usage at a node and minimum information maintained at a node as required by the data-location anonymity requirement. This is achieved through efficient query routing algorithms and optimization strategies over UQDT. In particular, our techniques use a judicious combination of (i) overlaying multiple logical trees over the same physical network, each with a distinct root, and arranging for queries to be channeled in parallel through distinct trees, and (ii) maintaining limited selectivity information about advertised keywords to inform the query routing strategy.

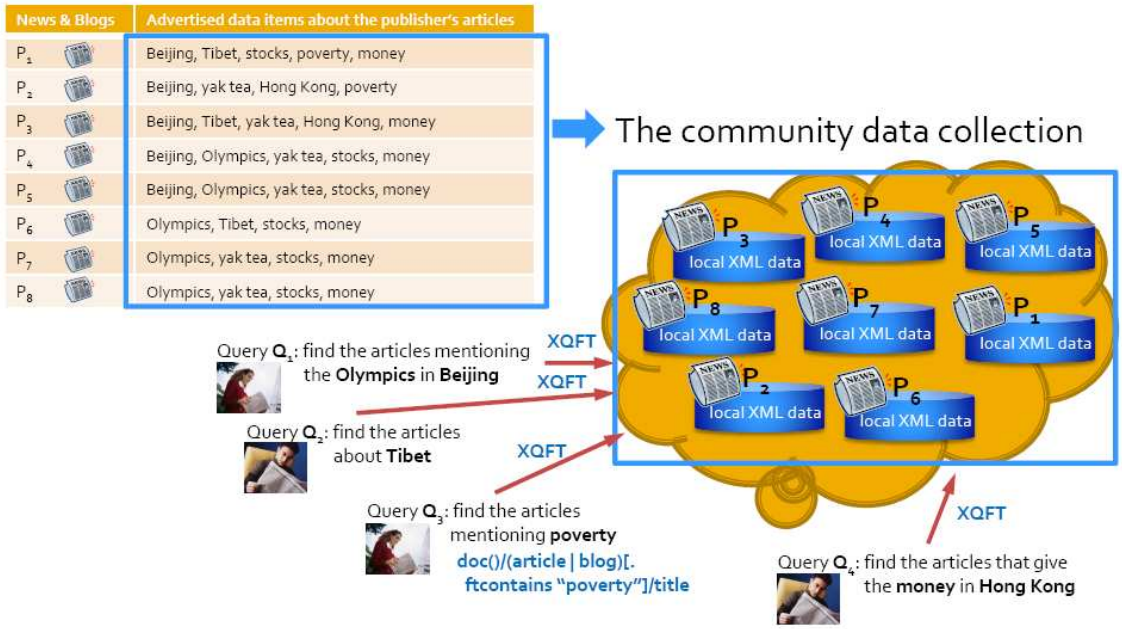


Figure 1.3: The community data: publishers advertise terms about their local data store, which is accessible via an XML-based full-text search interface

In addition, publishers may not want to reveal publicly the entire contents of their local store. Therefore, to protect their contents, we consider a model where publishers can specify what to expose by default as the community data. Publishers can advertise descriptive keywords about documents they publish and store at their own local repositories, as show in Figure 1.3. For instance, publisher P_1 contains

documents, stored in its data store, mentioning *Beijing, Tibet, stocks, poverty*, and *money*. As a result, our distributed index builds on these advertised keywords in the community and builds upon them to maintain the node summaries. The query forwarding infrastructure matches advertised keywords from the index with keywords mentioned in the query (usually based on a conjunction of query keywords) in order to discover relevant publishers that might contain documents of interest as shown in Figure 1.2. Since the local stores are not publicly direct available, the advantage is that these publishers can release their data to queriers only by giving their own consent explicitly.

1.3.2 Query Evaluation at Publishers

The query semantics when an XML full-text query Q is posed in the community, is to iterate through all the documents at the peer, test the full Q, and return to the consumer only those matching documents with Q. The final result consists of the union of the local results received from all individual relevant publishers in the community.

As a consequence of query routing, when a query hits a node, Q is evaluated on that peer's local XML store as it is detailed in Chapters 3 and 4. This is true since during query routing on UQDT, Q is partially tested on its keyword-conjunctive conditions only, ignoring the rest of its complex composable full-text conditions that are part of XQFT.

In order to efficiently process the expressive class of XQFT queries locally at the source, we design, implement, and deploy an XQFT engine processor at each source. In particular, we propose a reference platform processor for the XQuery Full-Text standard specification. In Chapter 3, we introduce a general technique for implementing the XQFT standard on top of an existing XQuery processor. We identify performance challenges, possible solutions, and their interaction with existing XQuery implementations. For efficiency, the back-end text engine is build on top of a local index store that contains posting lists for the documents at the publisher.

Inspired by this fruitful experience, we have leveraged our platform for addressing another challenging problem – that of efficiently evaluating the XQFT

languages. Currently, various communities made several competing proposals for XML full-text querying and scoring methods, which include the upcoming XQFT standard proposal. So far, the diversity of these proposals ruled out a unified treatment of their evaluation and optimization, imposing custom query processing for each language. To address this limitation, we propose a novel logical framework for query evaluation and optimization of such expressive languages in Chapter 4. One of the main challenges is to capture and formalize their semantics. This requires navigation on the XML tree structure while simultaneously processing complex full-text predicates to match keywords against the associated text.

Still, such expressive languages do not come with a relational flavor. Indeed, the XQFT's semantics is specified in a functional language. As such, devising an optimization framework in the spirit of the well-known techniques developed by the database community is desired and challenging. Yet, in Chapter 4, we developed a logical formalization and a score-aware algebra of these languages. Our first breakthrough was the ability to express the semantics of XML full-text languages relationally. We provide an alternative, yet equivalent specification in terms of tuples of matches of the query keywords into documents. Thus, in the spirit of relational query processors [102], our physical query evaluation plans are based on a set of algebraic operators that manipulate the matches of keywords in the text, as well as the scores. This allows to express the intermediate results of XQFT expressions as nested relations and suggests a relational algebraic approach to XML full-text query plans. As a result, we develop an optimization framework that enables optimization of algebraic plans in presence of well-behaved scoring functions by a rewriting engine inspired by classical relational equivalence rewriters [101, 49, 50].

The main benefit of our approach is an improved evaluation of XML full-text languages. This is achieved through (i) a concise and rigorous query semantics specification by translation into our algebra, and (ii) a uniform treatment of their optimization regardless of the language. In particular, the framework enables the tried-and-true relational equivalence algebraic rewritings (i.e., join reordering, pushing selections, etc.) to speedup the query execution. By implementing an XQFT query processor based on this logical framework, we showed that our approach outperforms existing XQFT solutions. To conclude, our work on XQFT query evalua-

tion tries to bridge the gap between the two communities – databases and information retrieval – by combining techniques for navigation on the document structure and traditional relational tuple-at-a-time manipulation with keyword search and full-text predicate evaluation.

1.4 Thesis Contributions

This dissertation improves on the state-of-the-art in XML information retrieval and distributed information systems. We propose and demonstrate the benefits of an integrated approach to build an efficient infrastructure that empowers information publishers to join free (e.g., democratic) communities and query their global data collection in an ad-hoc fashion using expressive queries over semi-structured data (e.g., XML full-text queries). This work is a step forward to successfully bridging the gap between databases and information retrieval communities.

In particular, to support democratic search over distributed XML peers in online communities with powerful expressive and composable queries, we make a number of contributions including the following:

- We propose a privacy-preserving enabling infrastructure in which data resides only with the publishers owning it. The infrastructure disseminates rich-expressive queries to publishers, who answer them at their own discretion. Moreover, the way in which publishers advertise their data, in order to receive relevant queries, is designed to prevent any third party from pinpointing which publisher advertises what data (without extensively colluding with or attacking community members).

- Given the virtual nature of the global data, we address the challenging problem of efficiently disseminating queries to publishers in the community that contain data matching a specified query.

  We propose a distributed index structure, UQDT, that is organized as a union of query dissemination trees (QDT's), and realized on an overlay network infrastructure. Each QDT has data publishers as its leaf nodes, and overlay

network nodes as its internal nodes; each QDT internal node maintains a summary of the data advertised by publishers in its subtree.

- We present algorithms that use the UQDT for routing queries to publishers efficiently, making effective use of the advertised data summaries maintained by the QDT. While a single QDT suffices in principle to route queries, this results in congestion at the upper levels of the QDT, severely limiting the throughput of the overall index structure, and making it potentially vulnerable to a Denial of Service attack. We show how UQDT can achieve load balancing and throughput maximization for a rich-expressive and diverse query workload.

  To show the effectiveness of our distributed access methods, we experimentally evaluate *UDT* design trade-offs through extensive simulations. We demonstrate that (i) one can statistically identify a near-optimum number of QDT's for any specified QDT topology, which maximizes throughput by preventing any overlay network node from becoming a bottleneck, and (ii) maintaining selectivity information about a limited number of popular advertised keywords (2-3%) achives considerable gains over a random routing strategy, and is almost as good as a fully informed routing strategy (i.e., 100% state).

- We present a general technique for implementing XQuery Full-Text specification using an existing XQuery processor. We describe GalaTex [61, 62], the first complete and conformance reference implementation of XQFT. In Chapter 3, we identify performance challenges, possible solutions and their interaction with existing XQuery implementations.

- Given the different XML full-text flavors of user queries, it is critical to enable their efficient evaluation on the local XML repository once they reach a relevant publisher.

  To this end, we introduce a formalization of XML full-text queries in terms of keyword pattern matches and present an algebraic foundation for such queries based on an algebra called XFT. Our algebra constructs and manipulates pattern matches as nested relations representing intermediate results of XQFT expressions. Therefore, most existing full-text languages can be expressed in

XFT, which enables a uniform treatment of their semantics specification, evaluation, and optimization problems. The XFT operators are freely composable, enabling query rewriting based on algebraic equivalences in the spirit of the relational algebra optimization. Finally, XFT can be seamlessly integrated with algebras for structured XML search, thereby enabling the optimization of queries which combine structured and full-text predicates.

- Finally, we demonstrate XTreeNet, an integrated approach to building a publishing system that combines distributed access methods to XML peers based on a simplified form of the query, followed by a full query evaluation (i.e., XQFT queries) at relevant publishers to return matching documents. We built a real distributed query dissemination platform and show, in Chapter 5, it can serve efficiently complex and expressive queries. This empowers information publishers to join democratic communities and empowers users to query the global data collection in an ad-hoc fashion.

1.5 Thesis Outline

The rest of the dissertation is organized as follows. Chapter 2 presents the efficient design of our UQDT infrastructure and the implementation of such distributed access methods to assist for query dissemination in online publishing systems. Chapter 3 and Chapter 4 present how to evaluate complex XML full-text queries locally, at the publisher. In particular, we describe in the first part a general way to build an XQuery Full-Text processor on top of an XQuery engine. In the second part, we show how to build a universal optimization solution and efficient algorithms for evaluation and optimization of XML full-text queries. Then, Chapter 5 describes the the XTreeNet system as a proof of concept for building a real-world distributed infrastructure for democratic publishing systems that can serve efficiently complex and expressive user queries. Finally, we conclude with a summary of the dissertation and some open challenges as future work in Chapter 6.

Chapter 2

Routing Queries using Distributed Access Methods

2.1 Motivation

During the last decade, the web has enabled unparalleled access to the vast amount of electronic data that is continually being created. Moreover, search engine technology has made it feasible to issue queries and locate web sites that contain data of interest to a user.

As the web evolves, two significant new trends are emerging that advocate for "democratization of information creation". First, write access to the web is becoming increasingly democratic as it is easier for a large number of users to create and publish data on a wide variety of topics; this is evident from the proliferation of blogs (e.g., Blogger [2], WordPress [21]), wikis (e.g., Wikipedia [20]), user-generated videos and photos, etc. Second, it is becoming easier to form web communities based on shared interests; this is evident in the considerably popularity of social networking sites like Facebook [4], Twitter [19], MySpace [14], and LinkedIn [12]. With the confluence of these two trends comes the natural desire to freely exchange data within the community – this includes making one's own data collection accessible to others within the community, and also be able to query, tag, and comment on the global collection that is the union of all local data collections of users within the community.

Understandably, users also want to maintain complete control over the access

to their own data. Recent events have shown that, in order to fully deliver on the promise of freely exchanging data and freedom of speech online, any community-supporting infrastructure needs to enforce the key requirement to preserve privacy of the association of publishers with personal intention and with potential sensitive or controversial information. That is, there should be no easy way for any third party authority to infer or to collude published data to reveal the identity of all publishers of documents on specific topics. Since this information can be further used to deny users access to certain publishers and their content or to deny selected publishers access online, the above requirement plays a paramount role in enabling freedom of speech online and therefore, in protecting users from censorship, harassment, and possible discrimination by third parties, be they of governmental, corporate or other nature.

In general, an effective such publishing system requires a variety of enforcement mechanisms to achieve democratic exchange of information, including user anonymity, encrypted channels, and data-location anonymity techniques. Whereas the first two mechanisms are well studied in censorship-resistant systems, they require dedicated trusted servers to encrypt and route the traffic through established anonymous tunnels. In practice, it is non-trivial to maintain and to protect these servers dynamically. We refer the reader to the related work section 2.8 for aspects related to these first two issues. Therefore, our infrastructure complements their resistance with another mechanism, called data-location anonymity (DLA), that preserves privacy of users in publishing communities in the sense that it provides anonymity of the association of users with certain specific information. In this paper, we focus on this third key aspect which is to prevent the query forwarding infrastructure from leaking information about which publishers are capable of answering a given query. Our proposal enforces that it is not easy for third parties to find out who published what information (i.e., data item) in order to prevent censorship, harassment, and discrimination of particular publishers and restrict access to information for consumers.

This privacy-preserving publishing requirement precludes some obvious approaches that reuse and build on existing centralized technologies, e.g., search engines, hosted online communities, etc. While these are designed to handle the large

number of potential publishers and the dynamic nature of published data, enabling a straightforward query access to the global data collection, the downside is that publishers are *disintermediated* from consumers by the central site:

- The central site has control over the visibility of publishers to user queries, and can effectively censor publishers by choosing to not index them.
- The central site has complete knowledge of all the information created by the publishers (in case they relinquish a copy of their data, as is usual with current search engines), or at least the topics advertised by the publishers. Even under the unrealistic assumption that this central site is trusted by the publishers, it is vulnerable to third-party censors [16, 15] and attackers.

For this reason, we advocate a decentralized approach in this thesis, where there is *no* central authority, and the global data collection is *virtual*. More specifically, we make the following contributions.

- We propose a distributed privacy-preserving enabling infrastructure in which data resides only with the publishers owning it. The infrastructure disseminates user queries to publishers, who answer them at their own discretion, under data-location anonymity constraints (i.e., prevent the query forwarding infrastructure from leaking information about which publishers are capable of answering a certain query). Moreover, the way in which publishers advertise their data, in order to receive relevant queries, is designed to prevent any third party from pinpointing which publisher advertises what data (without extensively colluding with or attacking community members).
- Given the virtual nature of the global data collection, we address the challenging problem of efficiently locating and disseminating queries to publishers in the community that contain data items matching a specified query.

  We propose a distributed index structure, UQDT, that is organized as a union of *query dissemination trees* (QDTs), and realized on an overlay (i.e., logical) network infrastructure. Each QDT has data publishers as its leaf nodes, and overlay network nodes as its internal nodes; each QDT internal node maintains a summary of the data advertised by publishers in its subtree. Unlike

Distributed Hash Tables (DHTs), no QDT node has complete knowledge of all the publishers that publish an advertised data item, thereby protecting publishers against information leakage even in case of attacks at network nodes or of colluding with a third-party censor.

- We present algorithms that use the UQDT for routing queries to publishers efficiently so that only those sources are accessed that can contribute to the query answer. Our algorithms follow the parent-child links present in a QDT and make effective use of the advertised data summaries maintained by the QDT internal nodes. While a single QDT suffices in principle to route queries, this results in congestion at the upper levels of the QDT, severely limiting the throughput of the overall index structure, and making it potentially vulnerable to a Denial of Service attack.

  We build on well known techniques for scalable dissemination trees and for "Russian Doll" search over sets [84]. We show how UQDT can achieve load balancing and throughput maximization for a workload W by a judicious combination of (i) Overlaying multiple QDTs over the network, each with a distinct root, and arranging for queries in W to be channeled in parallel through distinct QDTs, and (ii) Maintaining limited selectivity information about data items to help inform the routing strategy. To the best of our knowledge, there are no works that combine multiple trees for load balancing and hierarchical summaries for ad-hoc query routing in distributed systems.

- We experimentally evaluate UQDT design trade-offs through extensive simulations, using a real Wikipedia collection comprising about 1.1 million documents of total size 8.6GB. We demonstrate that UQDT can maximize throughput by preventing any overlay network node from becoming a bottleneck.

  To this end, we explore various QDT topologies (SCRIBE generated, as well as balanced structures), number of QDTs, and routing strategies (based on the selectivity information maintained), and show that (i) One can statically identify a near-optimal number of QDTs for any specified QDT topology, which maximizes throughput by preventing any overlay network node from becoming a bottleneck, and (ii) Maintaining selectivity information about a

limited number of popular data items ($2 - 3\%$) achieves considerable gains over a random routing strategy, and is almost as good as a "fully informed" routing strategy.

Chapter Outline. The remainder of this chapter is organized as follows. We start with an overview of our proposed framework and the space of design trade-offs in Section 2.2. Section 2.4 describes our implementation choices. The experimental setup is presented in Section 2.6 and the results are in Section 2.7. We discuss related work in Section 2.8 and conclude in Section 2.9.

2.2 Overview of UQDT Framework

Data and Query Model. For the purpose of information discovery and flexible querying, we abstract information as collections of *data items*, where each data item is described by a set of *content descriptors (CDs)*. CDs are an abstraction of keywords, terms, or other atomic information units [99]. For instance, in information retrieval applications, data items are text documents, and the CDs are the terms appearing in them. In relational databases, collections are tables, data items tuples, and CDs are (attribute,value) pairs. Further examples are given in Section 2.6. Given a data item D, we denote its set of CDs with $\mathsf{cd}(D)$.

We consider queries expressed as sets of CDs, and denote the set of CDs of query Q with $\mathsf{cd}(Q)$. We say that data item D *matches* query Q if $\mathsf{cd}(Q) \subseteq \mathsf{cd}(D)$. Notice that the case of matching conjunctive keyword queries against text documents (the most common Information Retrieval operation) corresponds to the particular case in which CDs are keywords. Given a data collection $\mathcal{D}$, the *result* of Q on $\mathcal{D}$, denoted $Q(\mathcal{D})$, is the set of data items in $\mathcal{D}$ that match Q: $Q(\mathcal{D}) := \{D \in \mathcal{D} \mid D \text{ matches } Q\}$.

Communities of Data Publishers and Consumers. We consider communities of autonomous publishers, who join the community with their own locally stored data collection and make it available for querying. In return, they can query the *global collection* consisting of the union of all local collections.

We contrast two competing approaches to designing the infrastructure for such communities. At one extreme lies the centralized approach, where all data

from the publishers is collected at a single site (this is what all search engines and hosted online communities do). The advantage of this approach is that querying the global collection is straightforward. The downside however is that publishers are disintermediated from consumers by the central site, and hence they lose control over who accesses or is interested in their content. At the opposite extreme, there is no central authority in charge of deciding where individual data items should reside and to re-shuffle them accordingly. In this latter setting the global collection is *virtual*, i.e. it is not materialized at any central location. Instead, data resides only with the publishers owning it. The advantage is that publishers maintain complete control over who accesses their content, and how the content is "advertised" to the community. The challenge here is efficient query evaluation. Given our focus on providing community-enabling infrastructure for autonomous publishers, we adopt the decentralized approach.

The virtual nature of the global data collection raises the challenge of avoiding the naive broadcast of queries to all publishers. We say that a publisher is *relevant* to user query Q if one of its local data items matches Q. What is needed is a distributed index structure that supports sending Q to all publishers relevant to it while minimizing the number of irrelevant publishers reached by Q. In this thesis we propose such an index so that only those sites are accessed that can contribute to the query answer.

Our indexing solution targets a service-oriented logical network infrastructure, in which we distinguish two types of nodes. There are data *publisher nodes* (the community members) that provide data services and connect to the network via direct links to nodes at its edge. The data are indexed inside the network, which consists of a set of inter-connected and reconfigurable *router nodes*. These are responsible for routing queries to the relevant publishers.

While different queries might hit the same set of nodes, our goal is to balance the community search generated load at routers during query dissemination while preserving both low space usage of index at a node and minimum information maintained at a node as dictated by the data-location anonymity requirement

2.2.1 Query Dissemination Trees

It is natural to organize peers and to index community data in a distributed way such that the peers storing relevant data can be located by contacting as few hosts in the network as possible. A popular way used in most distributed stream-based processing, DNS resolution, and publish-subscribe systems is to use a hierarchical representation for its better performance and efficiency, simplicity of protocols and low maintenance cost. Similarly, we propose the organization of the internal nodes into a logical tree called a *Query Dissemination Tree (*QDT*)*.

The internal QDT nodes are routers, the publishers are leaves. Regardless of which querier initiates a query Q, Q is sent to the root of the QDT, whence it propagates down the tree to the publishers. This is similar to network multicasting [47] in the sense that the query dissemination starts at the root node and follows down via relay nodes to the group members. However, our intention is that, when Q reaches a node n with no publishers in its subtree that are relevant to Q, n prunes its subtree from the search, i.e. it does not forward Q to its children. This pruning saves the network traffic and processing at n's descendants. Indeed, the tree shape of the QDT index is an optimal way to connect the group publisher members to the dissemination point. This ensures an efficient utilization of the network resources as (i) the data packets do not traverse a link more than once and (ii) we can prune irrelevant subtrees during query dissemination.

One immediate technical difficulty associated with this goal is how to instrument the index to efficiently determine that none of n's descendant publishers are relevant to Q. Of course, it is infeasible to maintain at every node n the collection of all data items in n's subtree. This would be prohibitively wasteful in terms of space, and it would defeat the purpose of preserving privacy of publishers: it would require a publisher p to trust (the good intentions and security of) every router on the path leading to p from the root. This is an unrealistic prerequisite.

We adopt a solution in which publishers share only limited information with the routers. Publishers advertise the contents of their local store by declaring a set of CDs appearing in their local collection. Note that not all existing CDs need to be declared, especially if they pertain to private data items.

We present in two steps the way routers exploit this information.

In a first cut, we assume that it is feasible to store at every node n the set $\mathsf{cd}(n)$ of all CDs declared by publishers located in n's subtree (we revisit this assumption shortly). This assumption is supported by empirical evidence that, for real data sets, the overlap of CDs across data items in a collection is considerable, and the union of all CDs (with duplicate removal) is orders of magnitude smaller than the combined size of the collection. For instance, in Section 2.6 we describe a collection of 1.1 million Wikipedia articles of combined size 8.6 GB that has only 3.2 million distinct CDs.

Let us observe that the QDT is also (iii) an effective and efficient way to maintain (i.e., build and update) these $\mathsf{cd}(n)$ aggregates at nodes where generic information is aggregated at the top and more specific information at the bottom of the hierarchy.

Note also that when only $\mathsf{cd}(n)$ is stored at a router n, n does not know which CD appears in which publisher, nor which sets of CDs appear together in a data item. This offers publishers an added degree of protection against compromised routers or external attacks.

2.2.2 Query Routing in Single-QDT

In this setting, we consider the following simple query routing algorithm. Every query Q posed by a querier p is initially sent to the root of the QDT (in a message containing both Q and p's address). When a router node n receives the message, it forwards it in parallel to each of its children in QDT if and only if $\mathsf{cd}(Q) \subseteq \mathsf{cd}(n)$. When a publisher node is reached, it sends back to p the result of Q against its local collection. Note that when $\mathsf{cd}(Q) \not\subseteq \mathsf{cd}(n)$ holds, it is guaranteed that n's publisher descendants are irrelevant to Q. Therefore, the first-cut routing algorithm never prunes relevant publishers, thus ensuring that the final result of Q over the global collection is computed in full. In contrast, when $\mathsf{cd}(Q) \subseteq \mathsf{cd}(n)$ holds, it is not necessarily the case that at least one publisher in n's subtree is relevant to Q. This is because the CDs in $\mathsf{cd}(Q)$ may not be co-located in the same data item, or even at the same publisher. Therefore, the first-cut algorithm may forward queries unnecessarily, generating non-minimal traffic and processing load. This is a result of

the unavoidable trade-off between privacy and evaluation performance.

EXAMPLE 2.2.1. *Throughout this chapter we use the following running example. Consider a network of* 25 *nodes that integrates general news from* 8 *different newspaper web sites* $P_1, ..., P_8$ *(the remaining 17 nodes are routers). Figure 2.1a shows the CDs declared by each publisher (they are simple keywords). Consider also a query workload consisting of the four queries shown in Figure 2.1b. Without showing the actual documents, assume that for every query* Q *there is at least one newspaper web site that publishes a document matching* Q*.*

Publisher p	CDs declared by p, $\mathsf{cd}(p)$
P_1	Beijing, Tibet, stocks, poverty, money
P_2	Beijing, yak tea, Hong Kong, poverty
P_3	Beijing, Tibet, yak tea, Hong Kong, money
P_4	Beijing, Olympics, yak tea, stocks, money
P_5	Beijing, Olympics, yak tea, stocks, money
P_6	Olympics, Tibet, stocks, money
P_7	Olympics, yak tea, stocks, money
P_8	Olympics, yak tea, stocks, money

(a) Publishers' declared CDs.

Query Q	$\mathsf{cd}(Q)$
Q_1	Beijing, Olympics
Q_2	Tibet

Query Q	$\mathsf{cd}(Q)$
Q_3	poverty
Q_4	Hong Kong, money

(b) Query workload.

Figure 2.1: Running Example Setup

*Assume for now that the routers and publishers are organized in the single-*QDT *configuration* QDT_1*, shown in Figure 2.3a. The router nodes are identified by their pre-order traversal rank. For simplicity, we assume that it is feasible for each node* n *to store all CDs declared by the publishers in its subtree,* $\mathsf{cd}(n)$*. The corresponding CD sets are shown in Table 2.1.*

For simplicity sake, let us consider in this example that every node can process exactly one query per time unit. If all queries in the workload are issued simultaneously at time 0 *and processed in the order* Q_1 *to* Q_4*, then Table 2.2 shows their dissemination according to the first-cut routing algorithm. For example, regardless*

Table 2.1: CDs Stored at Routers in Single-QDT Case

Node n	CD summary $\mathsf{cd}(n)$
4	Beijing, Tibet, stocks, poverty, money
6	Beijing, yak tea, Hong Kong, poverty
3, 2	Beijing, Tibet, stocks, poverty, money, yak tea, Hong Kong
10	Beijing, Tibet, yak tea, Hong Kong, money
9, 8	Beijing, Tibet, yak tea, Hong Kong, money, Olympics, stocks
14	Beijing, Olympics, yak tea, stocks, money
18, 17	Olympics, Tibet, stocks, money
21, 20	Olympics, yak tea, stocks, money
24, 23	Olympics, yak tea, stocks, money
16	Olympics, Tibet, stocks, money, yak tea
13	Beijing, Olympics, Tibet, stocks, money, yak tea
1	Beijing, Tibet, Olympics, yak tea, stocks, money, poverty, Hong Kong

of the issuing node, query Q_3 is disseminated in QDT_1 starting from the root node (node 1), which is congested and can only process Q_3 at time unit 3. Because poverty *is contained in* $\mathsf{cd}(1)$*, Q_3 is forwarded to all of node 1's children, in this case to nodes 2, 8 and 13, where the dissemination continues recursively. Since* poverty *does not appear in the CD sets of nodes 8 and 13, their subtrees are pruned (i.e. nodes 8 and 13 do not forward Q_3 to their children). However, node 2's CD set does match Q_3 and the query is routed down to node 3 at time unit 5, then to nodes 4 and 6 at time unit 6. Both these nodes have a match and Q_3 reaches the publisher nodes P_1 and P_2 at time unit 7. Each of the two publishers runs Q_3 on its local collection and sends the result back to the issuing node.* ◇

2.2.3 CD Set Summaries

We now revisit the assumption that all CDs in $\mathsf{cd}(n)$ are stored with every router n. We address the case when $\mathsf{cd}(n)$ is larger than can be comfortably stored at a router n with available memory of size M. To this end, we observe that we do not necessarily need to keep the exact set $\mathsf{cd}(n)$. Instead, it suffices to store a *node summary* thereof at node n. This is a data structure smm^M that fits in memory of size M and implements a boolean method contains such that for any set S of CDs,

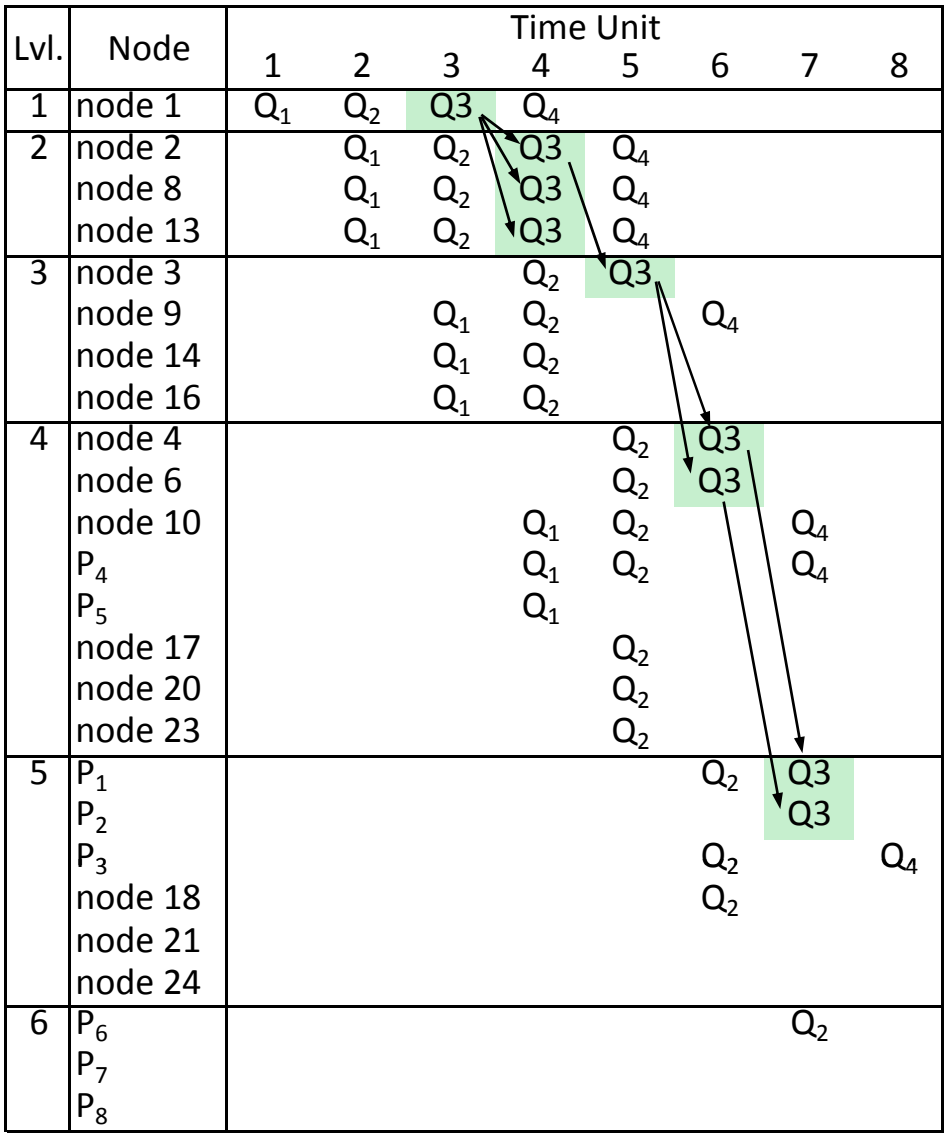


Figure 2.2: Query Dissemination in Single-QDT Configuration

the call $n.\mathsf{smm}^M.\mathsf{contains}(S)$ approximates the actual test $S \subseteq \mathsf{cd}(n)$. We obtain the final version of our routing algorithm by replacing in the above first cut every containment test with a call to the summary's contains method. To preserve in the final version the desirable properties of the first-cut routing algorithm, we require $n.\mathsf{smm}^M$ to satisfy the following. If $S \not\subseteq \mathsf{cd}(n)$ but $n.\mathsf{smm}^M.\mathsf{contains}(S) = true$, we say that $n.\mathsf{smm}^M$ gives a *false positive* on S. $n.\mathsf{smm}^M$ gives a *false negative* if $S \subseteq \mathsf{cd}(n)$ yet $n.\mathsf{smm}^M.\mathsf{contains}(S) = false$. Regardless of the memory size M, we disallow false negatives, since these would lead to incomplete computation of the

query result, pruning potentially relevant subtrees. In contrast, false positive do not affect the correctness of query evaluation, but impact its efficiency, as they result in unnecessary query forwarding. As seen above, unnecessary forwarding to publishers that eventually turn out to be irrelevant is not entirely avoidable even when $\mathsf{cd}(n)$ is stored exactly. Therefore, it is not crucial to guarantee the absence of false positives, but it is desirable to minimize their frequency, which should ideally decrease as the the available memory size M increases. We summarize the requirements on $n.\mathsf{smm}^M$ in the following list.

(a) $n.\mathsf{smm}^M$ fits in memory of size M;

(b) the call $n.\mathsf{smm}^M.\mathsf{contains}(S)$ is efficient, i.e. runs in time independent of the size of $\mathsf{cd}(n)$ and only linear in that of S;

(c) for every memory size M, $n.\mathsf{smm}^M$ gives no false negatives (i.e. for every S, $n.\mathsf{smm}^M.\mathsf{contains}(S) = false$ only if $S \not\subseteq \mathsf{cd}(n)$); and

(d) if $M_1 < M_2$, then the frequency of false positives is lower for smm^{M_2} than for smm^{M_1} (where the two data structures summarize the same set of CDs).

In Section 2.4, we present one concrete summary implementation based on *Counting Bloom Filters* [39, 70] of ssize M, proving that it satisfies our requirements. The bloom filter representation is a good choice due to their well-known properties including compactness and probabilistic set membership of CDs (i.e., no false negatives and good control over false positives rate). However, any alternative implementation qualifies.

2.2.4 Throughput Maximization

We have so far confined our discussion to the routing of a single query through the network. We next extend our solution to handle query workloads (sets of queries).

We start by observing that the arrival of a query at node n triggers measurable computation effort pertaining both to the processing of the query (lookup in the summary and evaluation over local collection if present) and to its forwarding to n's children. This limits the number of queries passing through n per time unit and can

lead to congestion. Since queries pruned at upper levels in the tree never reach the lower levels, the fraction of any workload W reaching node n is a subset (often strict) of the fraction reaching its ancestors. In particular, the root becomes a bottleneck since it is reached by all of W. In contrast, edge routers at the leaves are reached by relatively small fractions of W and may not be heavily utilized.

EXAMPLE 2.2.2. *Revisiting Figure 2.2, observe that the number of query messages reaching the nodes is significantly skewed among the tree levels, and ultimately among the nodes, decreasing from the root to the leaves. Because all queries touch the top 2 levels, their nodes receive 4 messages each, while nodes on the lower levels receive 0, 1, 2 or 3 messages. Overall, it takes a total of 8 time units to disseminate all queries, of which the root alone introduces a delay of 4 time units, while nodes 21 and 24 remain idle.* ◇

We propose to alleviate congestion at the upper levels of the QDT by spreading the load more uniformly across the nodes. Currently, there are two main solutions to achieve this. One class of algorithms replicate data (or indices of it) redundantly at the router nodes. Thus, each router can initiate to answer queries. Nevertheless, this incurs increased updates cost as well additional space cost to store all replicas which is inappropriate with our initial set of goals. In contrast, we propose to partition the global data collection and interconnect the publishers for each partition block in a different overlay. We show next how this technique alleviates congestion while still preserving the space usage at routers.

Therefore, our solution consists in overlaying multiple QDTs over the network, each with a distinct root, and arranging for various fractions of W to be channeled in parallel through distinct QDTs. Since all QDTs are supported by the same underlying logical network, a network node n participates in several QDTs, receiving and forwarding queries via each of them. Balancing the load involves arranging for the distribution of levels associated with n to be (as close as possible to) uniform across the set of all QDTs. For example, the fact that n receives a high fraction of the queries flowing through QDT_1 because it resides on an upper QDT_1 level, is compensated by n being reached by only a small fraction of the queries flowing through QDT_2, where it resides on a lower level.

The goal of splitting the query workload into fractions that flow through distinct QDTs raises two fundamental technical obstacles we need to overcome.

The first pertains to controlling memory consumption at the router nodes. If a node n participates in multiple QDTs, it must maintain separate summaries for each of its subtrees. A key requirement is

(e) the total space used by the union of all summaries associated with n should not exceed the space used by n's summary in the single-QDT configuration.

We satisfy this requirement by arranging for each of n's summaries to pertain to *disjoint* CD sets. To this end, we partition the space of all possible CDs into a number of k disjoint blocks $\mathcal{P} = \{B_i\}_{1 \leq i \leq k}$. (We discuss shortly what considerations go into picking the value of k, and we describe in Section 2.4 how the partitioning is achieved in practice.) We call each B_i a *CD block*. We assign to each CD block its own QDT, obtaining a family

$$UQDT = \{QDT_i\}_{1 \leq i \leq k}$$

The second problem is the preservation of the query semantics. That is, we need to ensure that, by being routed only through a single QDT, a query is guaranteed not to miss any relevant publishers. We achieve this soundness property by requiring each QDT to satisfy the following:

(††) QDT_i contains as leaves all publishers whose local data collection has at least one CD in common with B_i.

We defer to Section 2.4 the discussion on how the internal nodes of each QDT_i are organized.

2.2.5 Query Routing when Multiple QDTs

For every query Q, we pick the QDT to send it to as follows. The partition $\mathcal{P}$ induces a partition $\mathcal{P}_Q = \{Q_j\}_{1 \leq j \leq m}$ on $\mathsf{cd}(Q)$, such that for each $Q_j \in \mathcal{P}_Q$ there is $B_i \in \mathcal{P}$ with $Q_j = \mathsf{cd}(Q) \cap B_i$. We call each such Q_j a *query block* and we say

that the CD block B_i *corresponds* to Q_j. Note that by definition each query block corresponds to precisely one CD block, which in turn corresponds by construction to precisely one QDT. Given a query block $Q_j \in \mathcal{P}_Q$, we can therefore refer to "the" corresponding QDT, and denote it with $QDT(Q_j)$.

In general, Q has $1 \leq m \leq |\mathsf{cd}(Q)|$ query blocks, with corresponding QDTs $\mathsf{qdt}(Q_1), \ldots, \mathsf{qdt}(Q_m)$. For routing Q, we only pick one of these QDTs, say $\mathsf{qdt}(Q_j)$. Regardless of how this pick is taken, we send to the root of this QDT a message containing three components: (Q_j, Q, p), where p is the address of the initiating querier. $\mathsf{qdt}(Q_j)$ routes this message as described above in the single-QDT case, with only three minor refinements:

- since every internal node n can participate in various QDTs, n stores one summary smm_T^M per QDT T;
- n uses Q_j for routing in $\mathsf{qdt}(Q_j)$ (i.e. for lookup into the summary $n.\mathsf{smm}^M_{\mathsf{qdt}(Q_j)}$); and
- leaf nodes use Q for evaluation against their local data collection.

We summarize our discussion so far in the pseudo-code of Algorithm 2.1 and Algorithm 2.2 below.

Algorithm 2.1 $\mathsf{eval}(Q, p, \mathcal{P})$

Require: query Q, address of its initiator p, partition $\mathcal{P}$ of CD space
Ensure: pick the proper QDT to route the Q on
1: find $\mathcal{P}_Q = \{Q_1, \ldots, Q_m\}$ induced by $\mathcal{P}$;
2: pick $j \in \{1, \ldots, m\}$;
3: $\mathsf{route}(Q_j, Q, p,$ root of $\mathsf{qdt}(Q_j), \mathsf{qdt}(Q_j))$;

EXAMPLE 2.2.3. *Example 2.2.2 shows how the congestion appears inevitably in the upper levels of the dissemination tree. Here, we show how congestion can be alleviated by using multiple* QDT *overlays over the same nodes.*

We consider a configuration of 4 QDT*s, each corresponding to a block in the CD space partition $\mathcal{P}$. $\mathcal{P}$ is shown in Table 2.2.*

Algorithm 2.2 route(Q_j, Q, p, n, T)

Require: query block Q_j, query Q, address of its initiator p, reference to node n, reference to QDT T

Ensure: route query Q on QDT T starting from node n

```
1: if n is leaf then
2:     run Q over n's local collection;
3:     send result (if non-empty) to p
4:     return;
5: end if
6: if n.smm_T^M.contains(Q_j) = false then
7:     return;
8: end if
9: for each child c of n in T do in parallel do
10:    route(Q_j, Q, p, c, T);
11: end for
```

Table 2.2: Blocks of the 4-Partition

Block	CDs
B_1	Beijing, Olympics
B_2	Tibet, yak tea
B_3	Hong Kong, stocks
B_4	poverty, money

In general, internal nodes can be connected in any configuration at the network overlay layer. Figure 2.3 depicts 4 possible QDT*s, one per CD space partition block.*

Table 2.3 shows the CD summaries maintained at every router. Since a router appears in multiple QDT*s, it actually manages a set of summaries. For example, node 3 has one summary corresponding to nodes P_1 and P_2 which are all its publisher descendants in QDT_1, a summary for all publishers in QDT_2, one for P_8 in QDT_3, and a fourth summary for P_2 in QDT_4. To simplify presentation of this small example, we assume that each summary stores the exact set of CDs rather than its approximation.*

Table 2.4 presents the routing diagram in the 4-partition over time, assuming queries $Q_1 \ldots Q_4$ are issued simultaneously at time 0.

Query Q_1 is a conjunctive query both of whose CDs fall in the first partition block B_1. The only routing choice is hence the tree corresponding to B_1, namely QDT_1 shown in Figure 2.3a. Since $\mathcal{P}$'s blocks are disjoint, single-conjunct queries

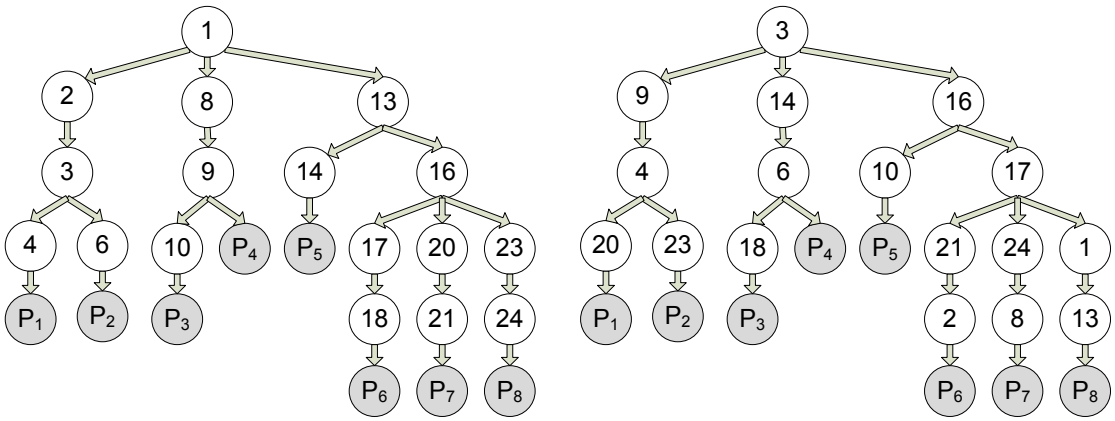
Table 2.3: CD Summaries in the 4-QDT Configuration

Node	Tree	Data summary
4, 6, 3, 2, 10	QDT_1	Beijing
9, 8, 14, 13, 1		Beijing, Olympics
18, 17, 21, 20, 24, 23, 16		Olympics
20, 2, 21	QDT_2	Tibet
23, 10, 8, 24, 13, 1		yak tea
4, 9, 18, 6, 14, 17, 16, 3		Tibet, yak tea
24, 18, 9, 8, 14, 13, 16, 3, 21, 17	QDT_3	stocks
1, 2		Hong Kong
20, 6, 23, 10, 4		Hong Kong, stocks
9, 1, 18, 2, 6, 14, 10, 16, 17, 4, 8, 21	QDT_4	money
3		poverty
13, 24, 23, 20		poverty, money

also have only one routing choice. For instance, Q_2 and Q_3 are routed using QDT_2 in Figure 2.3b, respectively QDT_4 in Figure 2.3d. Q_3's routing on QDT_4 by CD poverty *is highlighted in Table 2.4. It starts from the root (node 20) at time unit 1. Since* poverty *is contained in the node's summary, Q_3 is forwarded to nodes 23, 18 and 21. Only node 23 has a summary match and it forwards Q_3 further down to node 24, which recursively routes the query to nodes 13 and 3 at time unit 4. Both these nodes have a summary match, and publishers P_1 and P_2 receive Q_3 at time unit 5. However, because of processing contention at P_1 (P_1 is busy processing Q_2 at time unit 5), Q_3 will be served by P_1 at time unit 6 while P_2 serves it upon receipt at time unit 5. Both publishers contain matching documents and send them back to the query issuer.*

In contrast, query Q_4 intersects CD blocks B_3 and B_4, which induce two query blocks: $\mathcal{P}_{Q_4} = \{\{\text{Hong Kong}\}, \{\text{money}\}\}$. This offers two routing alternatives: either by using CD Hong Kong *on QDT_3, or by using* money *on QDT_4. In the diagram, we assume that QDT_3 was picked. When the subquery hits publishers P_2 and P_3 the full query Q_4 is tested on the local store (only P_3 has a match for both CDs of Q_4).*

*Comparing with Example 2.2.1, notice that the 4-*QDT *configuration outperforms the single-*QDT *case: the former takes 6 time units to complete the dissemination, while the latter needs 8. The improved throughput is due to better load*



(a) QDT_1 for B_1.

(b) QDT_2 for B_2.

(c) QDT_3 for B_3.

(d) QDT_4 for B_4.

Figure 2.3: Query Distribution Trees for the 4-Partition

balance: contrast the behavior of routers 21 and 24, which remain completely idle in Table 2.2 but shoulder part of the dissemination task in Table 2.4.

*Finally, observe that the benefit of better node utilization outweighs the drawback of using query blocks for pruning, instead of the entire (and more selective) set of query CDs. Indeed, the 4-*QDT *configuration wins despite its less aggressive pruning which leads to slightly more messages (50, as opposed to 46 for one* QDT*).*
◇

It is easy to check that property (††) on Page 32 implies the soundness of our query evaluation algorithm:

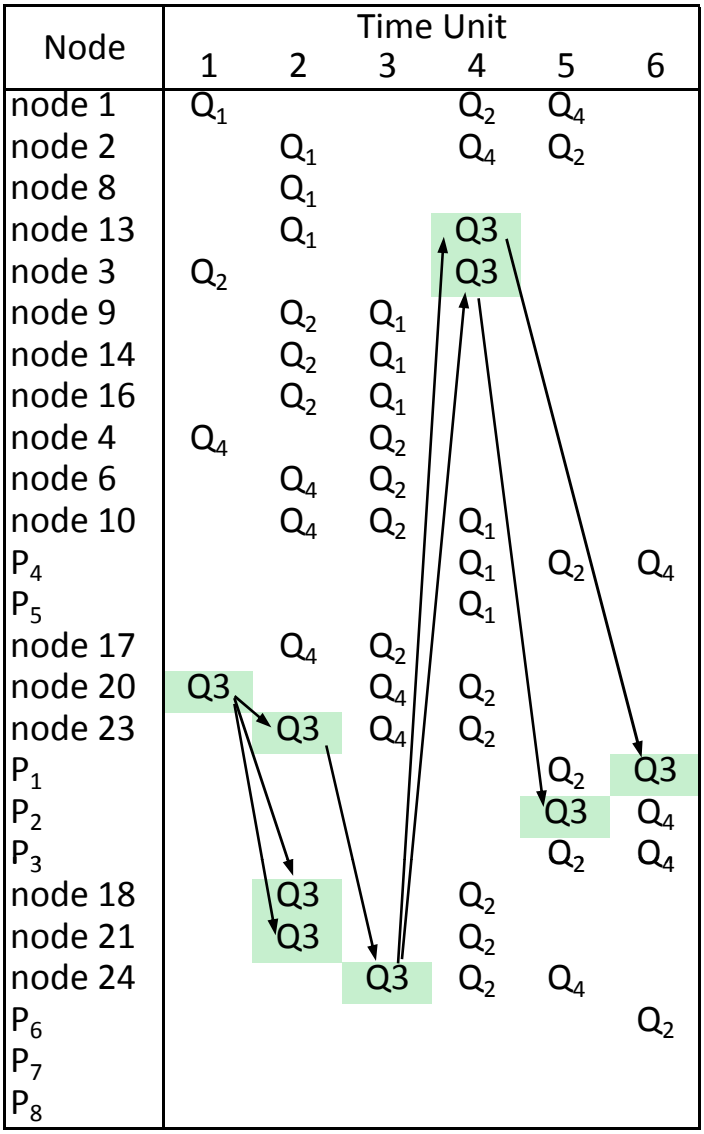


Figure 2.4: Query Dissemination in 4-QDT Configuration

Proposition 1. *For every query Q, partition $\mathcal{P}$, and every pick of j, Algorithm 2.1 correctly computes Q's answer.*

Obviously, for single-block queries there is no choice and the QDT is uniquely determined. However, in the general (and more likely) case of multiple-block queries, Proposition 1 uncovers an optimization opportunity: the judicious QDT choice (out of several equally sound alternatives) towards throughput maximization. We therefore need to treat the spectrum of possible routing strategies as an optimization dimension in its own right.

2.3 The UQDT Design Space Layout

We remark that the number k of blocks in the partition $\mathcal{P}$ of the CD space defines a spectrum of possible configurations of the same network, thus adding a new dimension to the optimization space. One extreme of this spectrum is the case $k = 1$, which we have discussed above as the single-QDT configuration. At the other extreme, we have the case in which each block of $\mathcal{P}$ is a singleton CD. We refer to it as the *per-CD* configuration. We argue next that neither of the extremes results in optimal throughput, and that the value of k is an optimization dimension we need to explore. Indeed, Example 2.2.1 and Example 2.2.3 show that the single-QDT configuration is certainly not optimal, being outperformed by a 4-QDT configuration for the given query load. At the same time, constructing too many QDTs is counter-productive, since the increase in k decreases the size of the query blocks, thus resulting in less selective lookups in each node's summary. This translates into less pruning, i.e. more query forwarding messages: the 4-QDT configuration in Example 2.2.3 generates 50 messages, as opposed to the 46 of the single-QDT configuration in Example 2.2.1. In conclusion, as k increases, we observe two opposite effects: an increase in load balancing potential, but also in the overall load (number of messages) in the network. An independent consideration that precludes extremely high values of k is that the maintenance of any overlay network involves a small, but non-zero control traffic overhead [47]. Maintaining too many QDTs would amplify this overhead. Finally, we observe that the per-CD configuration suffers from an additional problem: for every node n, any reasonable summary $n.\mathsf{smm}^M$ would have to contain at least one bit for the unique CD it summarizes, so the combined size of all summaries of n would amount to a prohibitively expensive value, linear in the number of all possible CDs.

In the next Section 2.4, we discuss the following issues not covered here, all of which have significant impact on query throughput: How can a partition $\mathcal{P}$ of the infinite space of all possible CDs be chosen and represented finitely (this includes determining the value for k)? How can $\mathcal{P}$ be used to efficiently determine $\mathcal{P}_Q$? How are the various QDTs corresponding to $\mathcal{P}$ organized for better throughput? How are the CD summaries smm_T^M at every node implemented and maintained to satisfy

requirements (a) through (e) above? How does the choice of QDT (the pick of j in Algorithm 2.1 – eval) impact throughput?

2.4 Our Approach

In Section 2.2, we have provided an overview of our proposed solution for query dissemination, identifying the dimensions of the space of possible implementations. As a proof of concept for the viability of the overall approach, we developed an actual implementation, described in this section and evaluated experimentally in Section 2.7.

We focus on the most flexible setting supported by our infrastructure, namely the case in which the service-oriented overlay network to which publishers connect is possibly owned by a separate entity distinct from the publishers. We therefore have a set of router nodes connected by an overlay network, and a set of publisher nodes who attach to this network to join the community.

2.4.1 QDT Topology

There are many possible topologies according to which we could organize the router nodes into a QDT. We investigate two approaches.

First, we take the pragmatic approach of "piggy-backing" on top of a mature overlay tree-building approach to disseminate messages to groups of nodes (also known as *multicast* groups). Since multicast overlay trees are constructed with a different goal than QDTs, it is not immediately clear that they are optimal for the purpose of query dissemination (though we show experimentally that we can "convert" them, achieving very good performance). However, one advantage of delegating the QDT construction to such off-the-shelf technology is that it is equipped to exploit information on the topology of the underlay network with minimal control overhead. Moreover, it maintains overlays dynamically, adapting to the change in underlay network conditions. One widely-used representative of this class of tools is Scribe [47]. We use as our platform the open-source SCRIBE implementation FreePastry version 2.0 beta 2 [17].

In addition, we consider home-grown QDTs built for the express purpose of balancing the forwarding effort among the routers. Since every router forwards a query to each of its children, the forwarding effort is linear in the node's fanout. This suggests constructing (nearly) balanced QDTs, with as little variation as possible in the node fanouts. We need to construct such trees ourselves, since Scribe does not guarantee balanced trees.

2.4.2 CD Summary Implementation

Regardless of the QDT topology, a key issue is the implementation of the CD summaries at every router node so as to satisfy the requirements (a) through (d) in Section 2.2. We implement a summary smm^M as a *Counting Bloom Filter* [70] of size M. That is, a data structure consisting of a vector of M counters, thus satisfying smm^M requirement (a). The vector is accompanied by l hash functions $\{h_i\}_{1 \le i \le l}$ from CDs to the set of integers corresponding to positions in the vector, $\{0, \ldots, M-1\}$. Every CD c corresponds via the hash functions to up to l indexes in V. We denote the set of these indexes with $\mathsf{ind}(\{c\})$, defined as $\mathsf{ind}(\{c\}) = \{h_i(c) \mid i \in \{1, \ldots, l\}\}$. Given a set S of CDs, we associate to S the set $\mathsf{ind}(S) := \bigcup_{c \in S} \mathsf{ind}(\{c\})$. To insert a CD c into smm^M, we simply increment all counters located at the positions in $\mathsf{ind}(\{c\})$. When looking up CD set S,

$$\mathsf{smm}^M.\mathsf{contains}(S) = \text{ true iff } \bigwedge_{i \in \mathsf{ind}(S)} V[i] > 0.$$

This immediately implies that smm^M cannot yield false negatives, since if the CDs in S are previously inserted in the summary, all the relevant counters are non-zero, and method $\mathsf{contains}$ must return *true*. Therefore, smm^M satisfies requirement (c).

Notice that the lookup of CD set S requires hashing each member $c \in S$ and accessing the vector at the l positions given by the hash functions on c. Since l is a constant, we obtain lookup time linear in $|S|$, thus satisfying requirement (b). We mention an optimization that minimizes the overall lookup effort for the CDs of a query block Q_j. Given a QDT T, all router nodes n in T implement their summary $n.\mathsf{smm}_T^M$ using the same l hash functions, and vectors of the same length

M. Consequently, there is no need to re-hash all CDs in Q_j at every node, as the result will be the same. Instead, nodes forward to their children in T the set $\mathsf{ind}(Q_j)$, computed once and for all at the root of T.

Requirement (d) follows from a celebrated property of Bloom filters, namely that if the l hash functions are independent, then the probability of false positives decreases monotonically with increasing M [70]. There are well-known techniques for constructing l independent hash functions for any l, for instance by linear combinations of only two generating hash functions [89]. We adopt this solution here. The generators can be picked from among several well-researched specimens. We use the SHA1 [69] cryptographic algorithm (that hashes strings of arbitrary size into 160-bit vectors) because it is very fast to compute and yields good distribution of the hashed values.

EXAMPLE 2.4.1. *As reported in Section 2.6, we will consider a global data collection of* 8.6*GB, featuring* 3.2 *million distinct CDs. Fixing the false positive rate at* 10^{-2}*, it follows from the formula in [70] that the optimum number of hash functions is* $l = 7$ *when the size of the Bloom filter at every router (assuming a single-*QDT *configuration and counters of size 1 bit) is* $M = 3.6$ *MB, which represents only* 0.044% *of the global collection size. For larger counter sizes, the false positive rate is even lower. For* k QDT*s, the global memory consumption per node stays the same, since the* k *Bloom filters at every node summarize disjoint sets of CDs. Each Bloom filter has size* $3.6/k$ *MB, and the same error rate of* 10^{-2}*.* ◇

2.4.3 QDT Maintenance

When a publisher p joins the community, it declares a set $\mathsf{cd}(p)$ of CDs it is willing to answer queries about. Recall from Section 2.2 that, to preserve soundness of query evaluation, we must satisfy property (††) from page 32. To this end, we determine (as described shortly) all the CD blocks with non-empty intersection with $\mathsf{cd}(p)$, which in turn lets us identify all QDTs that p must join. The act of joining a given QDT is taken care of by Scribe (or a traditional multicast join as with IP-multicast), which identifies the router node that will become the new publisher's parent. Once the publisher is added to QDT T, the CD summaries of all its ances-

tors in T are updated by inserting $\mathsf{cd}(p)$ into them. This insertion is implemented by simply obtaining once and for all the set of indexes $\mathsf{ind}(\mathsf{cd}(p))$, which is then passed bottom-up from p to T's root, so that every router on the way can increment its corresponding Bloom filter counters. When p leaves a QDT T, the index set $\mathsf{ind}(\mathsf{cd}(p))$ is also sent bottom up to p's ancestors in T, each decrementing the corresponding counters. The case when an existing publisher p changes its list $\mathsf{cd}(p)$ of declared CDs leads to the propagation of similar counter increment and decrement operations.

2.4.4 Partitioning the CD Space

An important issue we need to address is how to represent the partition $\mathcal{P}$ of the CD space finitely, and how to efficiently determine which block a given CD belongs to. As described above, we need this test to quickly identify the QDTs a new publisher must join. Moreover, the same test is required to compute the induced partition $\mathcal{P}_Q$ of a query Q, in order to identify the QDT candidates for routing Q. We describe here our solution assuming that we have already established the number k of blocks in $\mathcal{P}$ (we discuss below how we determine k with an eye on load balancing). Given k, we implement $\mathcal{P}$ simply as a hash function $h_{\mathcal{P}}$ from CDs to the set $\{1, \ldots, k\}$, where $h_{\mathcal{P}}$ distributes CDs uniformly over its range. Then each block $B_i \in \mathcal{P}$ consists of all CDs mapped by $h_{\mathcal{P}}$ to i: $B_i := \{d \mid d \text{ is a CD }, \ h_{\mathcal{P}}(d) = i\}$. Of course, each CD block is potentially infinite so we never really materialize it. Indeed, we don't need to: all we need is to quickly determine, given a CD d, which CD block it belongs to. This operation is implemented as a constant-time invocation of $h_{\mathcal{P}}(d)$.

2.4.5 Load Balancing

The way we determine the number k of QDT trees, as well as their actual construction, are motivated by the goal of spreading the load evenly across router nodes. For the following discussion, we denote with N_r the number of router nodes in the service provider's overlay network, and with N_p the number of publisher nodes. Since in any QDT T, every router node is reached by a larger fraction of the query flow through T than its descendants in T, we need to ensure that for every router n, the distribution of QDT levels n resides at is close to being uniform. We adopt a

solution which is certainly not the only possible one, nor necessarily optimal, but it is easy to implement and (as proven experimentally in Section 2.7) it yields excellent performance. We start by constructing (using SCRIBE) a single QDT T_1 whose internal nodes are the N_r routers and whose leaves are the N_p publishers. SCRIBE tends to build trees of low height, in which the root has a significant fanout that dominates the fanouts of nodes in lower levels. The root and the children of the root receive by far the highest fraction of queries flowing through the tree, and are hence in most need of relief through load balancing.

Denoting with N_u the number of nodes on the top 2 upper levels in T_1 ($N_u =$ 1+ number of router children of the root), we construct

$$k = \lfloor \frac{N_r}{N_u} \rfloor$$

QDTs, $\{T_i\}_{1 \leq i \leq k}$. Each T_i is an isomorphic copy of T_1, whose nodes are obtained by keeping the same N_p leaves and only re-shuffling the N_r internal nodes as follows. To completely specify T_i, we need to specify how its N_r internal node positions are populated with the actual N_r routers. This specification can be formalized as a function a_i from the set of N_r routers to the set $\{0, \ldots, N_r - 1\}$ of positions in T_1. We adopt the convention that the *position* of node n corresponds to n's rank in the breadth-first, left-to-right traversal of T_1 (position 0 is the root). Let $\pi(n) := (n - N_u) \mod N_r$ be the right-to-left cyclic permutation with step N_u on $\{0, \ldots, N_r - 1\}$. If a_1 specifies the initial QDT T_1, then for each $1 < i \leq k$, we populate T_i by cyclically permuting with step N_u the nodes of T_1 a total of $i - 1$ times: $a_i := \pi^{i-1} \circ a_1$.

EXAMPLE 2.4.2. *In Example 2.2.1, there are $N_r = 17$ routers, and the root of the initial tree QDT$_1$ has three children, yielding $N_u = 4$. We compute $k = \lfloor \frac{17}{4} \rfloor = 4$ and construct the 4 trees in Figure 2.3. Notice that the trees in Figure 2.3(b), (c), (d) are obtained by cyclically permuting to the left by 4 steps the tree in Figure 2.3(a) once, twice, respectively three times.* ◇

It is easy to see that our method of determining the number of QDTs, as well as our method of populating them, ensures the following fairness property:

(∗) All routers appear precisely once in the top 2 levels of

any QDT.

Furthermore, the k level values associated to every router are distributed almost uniformly over all possible level values in T_1. For instance, in Figure 2.3, router 1 appears on levels $1, 4, 4, 3$.

Finally, note that building $k + 1$ QDTs actually degrades the load balance, because the additional cyclic permutation causes a "wrap-around" that returns some of the routers residing on the top two levels in T_1 to the top two levels of T_{k+1}, subjecting these routers to unfair load (since we use the floor function to determine k, the wrap-around is not necessarily complete). In general it follows that, to maximize balance, we want to use a number of QDTs that is a multiple of $\lfloor \frac{N_u}{N_r} \rfloor$. In Section 2.7, we validate this rule experimentally, also showing that choosing multiples higher than 1 is unnecessary: they do not improve load balance, while leading to higher control overhead.

2.4.6 Load Balancing with Query Workload

Let us note that query workloads are not always uniform, but in reality, they tend to be skewed at times and very dynamic in general. For instance, it might be that during the US 2008 presidential campaign keywords like "Obama" appear at the top of the searches, or it maybe very well about Olympics in Beijing, Tibet, Michael Jackson, etc. Although the scope of this thesis is not to address the issue of adaptation with the query workload, we address it briefly here.

We argue that the UQDT infrastructure can handle these situations with success. To make our case, note that the number of possible CDs advertised in the community is huge – tens of millions. On the other hand, the number of QDTs is relatively small (e.g., tens) as it can be seen in Section 2.4.5. So, each QDT is handling millions of CDs. Since the assignment of CDs to QDTs is done via a random hashing, a popular query keyword in one QDT is likely to be balanced out by a number of less popular query keywords in the same QDT. In order to have one QDT overloaded, either a single query keyword would need to be a significant fraction (e.g., 10%) of all queries, or the collection of most popular CDs all need to map to the same QDT, which is unlikely in practice.

2.4.7 Routing Strategies

We next discuss how a node n that initiates a query Q picks the QDT to route Q on. First, n uses the hash function $h_{\mathcal{P}}$ described above to compute $\mathcal{P}_Q = \{Q_j\}_{1 \leq j \leq m}$, which in turn determines the set of candidate QDTs $\{\mathtt{qdt}(Q_j)\}_{1 \leq j \leq m}$. If $m > 1$, n picks one of these candidates. We consider several alternatives for implementing this pick.

A simple solution is to choose $1 \leq j \leq m$ at random, in the hope that randomness avoids sending too many queries down the same QDT and thus alleviates congestion. We call this the *random* routing strategy.

DEFINITION 2.4.1. (**Random Routing Strategy**) In the random routing strategy, a node n starts the dissemination of query Q in the network using $\mathtt{qdt}(Q_j)$ where j is randomly picked such that $1 \leq j \leq m$. ◇

We also consider alternative strategies, all attempting to alleviate the effect we discussed in Section 2.2: as the number of QDTs increases, the selectivity of query blocks decreases (recall that, when routing Q through QDT $\mathtt{qdt}(Q_j)$, only the CDs in Q_j are looked up in the summaries). This results in increased overall query forwarding and processing in the network. To compensate for this effect, the routing strategy should ideally use the most selective query block Q_j for routing, as this results in the most aggressive pruning of $QDT(Q_j)$'s subtrees during Q's dissemination. We call the strategy assuming each publisher's access to this information the *fully-informed* routing strategy.

DEFINITION 2.4.2. (**Fully-Informed Routing Strategy**) In the fully-informed routing strategy, a node n starts the dissemination of query Q in the network using $\mathtt{qdt}(Q_j)$ where j corresponds to the most selective query block Q_j among the $\mathcal{P}_Q$ blocks of Q accounting for all published CDs. ◇

Identifying the most selective block of a query is a non-trivial task, because it requires determining the frequency of every CD in the global collection, and storing these global statistics (or making them otherwise accessible) at every publisher node. Assuming independence between the CDs, the publisher initiating Q computes the selectivity of a query block Q_j as the product of the individual frequencies of the CDs

in Q_j. Fully-informed routing is very expensive in terms of both space and traffic. Indeed, for large global collections, the number of CDs can be considerable. Moreover, space consumption is exacerbated by the fact that the frequency information must be stored with every potential initiator of a query. An even more serious problem is the traffic arising because the global collection is virtual: gathering and maintaining the appropriate statistics requires constant communication between nodes.

We therefore investigate a less ambitious strategy: instead of identifying the most selective query block for Q, its initiator p only tries to avoid using the least selective ones. It suffices to this end to maintain and store at each publisher a short list of the s least selective (most frequent) CDs in the global collection, with s a relatively small value ensuring small storage space and maintenance traffic consumption. Finding the overall top s most frequent CDs amounts to solving a distributed top-s problem, in particular the classical problem of top-s heavy hitters estimation in data streams [31, 95].

We implement a simple solution that exploits the already existing QDToverlays, employing them in a dual role as multicast (data dissemination) trees. With every CD they advertise, publishers declare its frequency in their local collection. Each node n maintains a list $n.L$ of length at most s entries, each containing a CD and its frequency. For non-root routers, the list gives the s most popular CDs across all their QDTsubtrees. For QDTroots and publishers, the list holds most popular s CDs across the global collection. Whenever a node n updates its list, it propagates the new list bottom-up along all QDTs n participates in. If n is a root, it propagates its list to the other $k-1$ roots. Whenever the root of a QDTT updates its list, it disseminates it top-down to all publishers in T.

When node n issues a query Q, it picks the QDT corresponding to Q's most selective block according to the information in $n.L$. Note that some query blocks may contain CDs not occurring in the $n.L$ list. These are treated as selective CDs, and blocks with the highest number of selective CDs are preferred. If multiple such query blocks exist, n breaks the tie by computing the selectivity of the conjunction of popular CDs in each block, using $n.L$. If this still leaves more than one candidate query block, one is picked at random. We call this strategy *partially-informed* routing, and observe that it leads to a spectrum of strategies parameterized by the size

of internal state reserved for the list of popular CDs. We use the term *x-informed routing* in short for partially-informed routing based on the list of the most popular $x\%$ of CDs.

DEFINITION 2.4.3. (**x-Informed Routing Strategy**) In the x-informed routing strategy, a node n starts the dissemination of query Q in the network using $\mathtt{qdt}(Q_j)$ where j corresponds to the most selective query block Q_j among the $\mathcal{P}_Q$ blocks of Q accounting for only $x\%$ most popular of the published CDs. ◇

Notice that 100-informed routing becomes fully-informed, and 0-informed routing degenerates to random routing. In Section 2.7, we show experimentally that, by keeping track of even very short lists, we observe performance very close to the fully-informed strategy, and much better than the random strategy.

EXAMPLE 2.4.3. *We revisit Example 2.2.3, explaining why query Q_4, which had two routing alternatives, was sent to QDT_3. To enable fully-informed or partially-informed routing, publishers maintain frequencies of (some of) the CDs in the global collection, which in our case include* money *(published by 7 publishers),* stocks *and* yak tea *(published by 6 publishers). Notice that* Hong Kong *is declared by only 2 publishers and hence more selective than* money*, which is why it is preferred by the fully-informed routing strategy. Since CD* Hong Kong *appears in block B_3, the corresponding tree QDT_3 is used. The same outcome is achieved for partially-informed routing, assuming for instance that publishers maintain only the 3 most popular CDs: the list includes CD* money*, signaling to Q_4's initiator to avoid routing by it.* ◇

Finally, when no selectivity information is available, we fall back on *heuristic* routing: simply direct Q to the QDT corresponding to one of Q's maximum-cardinality blocks, breaking ties with random picks. This strategy is based on the heuristic that higher numbers of conjuncts tend to yield higher selectivity.

2.5 Trust Model for UQDT

We discuss next the ability of the UQDT infrastructure to enable freely exchanging of information among the users of the community of interest.

DEFINITION 2.5.1. (**Data-location anonymity**) We define *data-location anonymity*, or DLA, as the ability to prevent leakage of information that allows for third parties to associate published data with publishers in community-based networks, for the scope of finding which publishers can answer to certain (possible sensitive) queries.

More formally, DLA hides from third parties the exact association mapping between publishers and their advertised CDs:

$$Publisher2CDs : publisher \longrightarrow CDs$$

In particular, we are interested in guaranteeing the following two DLA axioms:

- (A_1) Make it hard to find all publishers for a certain CD cd_x (or for a given set of CDs), or to find

$$\{p | p \in Publishers\ such\ that\ Publisher2CDs(p) = cd_x\}$$

- (A_2) Make it hard to find all CDs for a certain publisher p_x (or for a given set of publishers), or in short, to find

$$\{cd | cd \in Publisher2CDs(p_x)\}$$

◇

For instance, Figure 2.1a on pages 27 shows the mapping of publishers to their advertised CDs in that particular sample scenario. The intuition behind DLA is to confer the users of the community the sense of protection (e.g., not being vulnerable) to having their identity information associated with specific published or queried data online. DLA resistance is challenging because knowledge about who publishes or queries information, which is not desired by third parties, may lend itself to compromise the users' identity. We strongly believe that if sensitive community data is tracked back to the person publishing it, that is evidence against them and it can be subject to user censorship, harassment, or discrimination. By ensuring DLA, our main concern is to make the advertised community data not traceable back with

the real publishers.

Our UQDT solution is a DLA resistant query forwarding infrastructure in the sense that it protects all publishers that are likely to answer a specific query. In particular, we claim that

Claim 2.5.1. *The* UQDT *precludes third parties from learning the exact publisher–CD associations Publisher2CDs (see A_1 and A_2) without compromising a significant fraction of the router nodes or without issuing a large number of queries, which both might be infeasible or too costly to put in practice.*

2.5.1 Assumptions

Next, we state a common and realistic set of assumptions under which we consider the trust model for UQDT to be DLA resistant.

We expect that in an Internet-scale distributed setting, the number of routers n in the overlay is relatively large. Moreover, it is natural that the overlay consists of a diversified router infrastructure, i.e., the routers are administered by a multitude of distinct third parties and administrative domains (e.g., ISPs) some of which can not be trusted. Hence, no party controls more than a small fraction of the entire infrastructure. This can be arranged in many ways. One extreme is a pure P2P network where each router is also a publisher, owned by a distinct individual. This is a realistic assumption. For instance, imagine that a distributed publishing and query paradigm for Facebook [4] can connect millions of users worldwide. As a results of this architecture, we do not consider attacks that collude information across different administration domains.

We also assume that even though the QDT topology shape may be publicly known, no node knows the position of other nodes in the QDTs. In other words, each node has only local topology information, mainly its own position in the QDTs it participates in and the directly linked nodes to itself.

We assume trusted publishers in the sense that they report valid data summaries and evaluate correctly incoming XML full-text queries to release the proper set of answers according to their preference policy.

We do not consider passive attacks or traffic analysis attacks that can listen on

the network without compromising nodes. This is because simply passively listening cannot leak the nodes organization as UQDT and the configuration of the multiple QDT overlays required for an attacker to jump on the corresponding nodes and monitor a query dissemination process.

We exclude weak access control in which users can easily forge new credentials to join the community network (e.g., sybil attacks). Thus, we assume a strong access policy for joining the community that prevents users without credentials to join. For instance, users' credentials are given only for those that paid their membership subscription.

Whereas distributed denial-of-service attacks (DDOS) constitute a valid class that by overloading a site or a QDT decreases the data availability, it is not the main focus of this thesis. However, note that the effect of DOS attacks on QDT overlays is mitigated by the fact that for different multi-CD queries, even if they share a popular CD, they may be routed along different QDTs, thus spreading the load evenly across the network. Plus, as long as this load remains below the network's capacity, DOS will not occur. Repeated same-CD queries would be routed along the same path (e.g., same QDT), but are easy to spot and react to by blocking.

2.5.2 Analysis

We show next how the UQDT infrastructure enforces and meets our goal of DLA resistant requirement.

Let us note that UQDT does not carry information at the level of documents in the index, but at the level of the advertised data items, also known as CDs, describing the published documents. Hence, there is no disclosure of which publishers control a certain document. Moreover, the actual document retrieval from the publishers can be complemented to be done anonymously and over encrypted channels using techniques such as [67, 68] as discussed in related work Section2.8.

Also, let us note that axioms A_1 and A_2 lend themselves to two different attacker models. Under our assumptions and design requirements, we consider next the following viable attack models as follows:

- (A_2) issue queries in the community: one can send queries in the community

if it is a registered community member in order to infer information about a particular QDT. For instance, to find all patients of a publisher hospital p, an observer can probe p with queries of the form: "is cd a patient of p?"

- (A_1) compromise nodes by external observers: on the other hand, if the first option is not possible, one can compromise nodes and get control over their own content. For instance, to find all health medical records for patient cd from all publisher hospitals, or to find all patients suffering from sickness cd from all hospitals, one can compromise nodes and get access to their data summaries and internal data.

Of course, one can also combine the two above attack techniques. We analyze next the strength of DLA resistance of UQDT against these attack types.

Issue Queries in the Community

In the context of axiom A_2, a plausible attack scenario would be to have a registered user of a virtual health services community for instance, that wants to find out who are all the patients cd of a certain hospital p. Since, the user does not know how to compromise nodes or doing that is illegal, impossible, or just too expensive, the user can issue queries to probe p directly at a certain cost c per query. Queries are of the following form "is cd a patient of hospital p?" for all possible cds. Therefore, the overall cost of such an attack is $C = \sum_{all\ queries}(c)$ to disclose A_2.

We argue that the cost C of how many queries to send to find out the contents of publisher p depends on the size of the application domain. In principle, the user needs to try the following set of queries: all sets of c CDs (which are all desired CDs to expose), all sets of $c-1$ CDs, ..., all sets of 1 CD (which is each CD individually). However, most of the times, this attack strategy is not even feasible since the space of exploration and therefore, the number of required queries, is too large, thus making it too expensive. For instance, another attack scenario would be for a spammer to find out all email addresses of gmail.com using this method. Obviously, testing each email address is too expensive and makes it infeasible in practice.

Compromise Nodes

In order to cover all use cases, we study in details the following attack scenarios: on a random node (internal router node, edge router node, publisher node), on set of nodes, and on all nodes of the community.

One key aspect is that of protecting the identity of the publishers of any given CD. By only maintaining data summaries at every router, our design severely limits the amount of identification information that can be gained by compromising (i.e. colluding with, impersonating, hacking or subpoena-ing) any single router r.

First, no information is gained about publishers not located in r's QDT subtree(s).

Second, our proposed distributed index partitions the index space rather than replicate it over the nodes as explained in Section 2.2.4. In general, this has the advantage that compromising a random node can only leak its local data summary together with its local topology information about the QDTs it participates in. Unlike DHT systems in which a DHT node usually manages all the information about each data item, no QDT node has complete knowledge of all the publishers that advertise a data item or a CD, thereby preventing QDT nodes from disintermediating publishers.

Note that when only $\mathsf{cd}(r)$ information is stored at a router r, r does not know which CD appears in which publisher, nor which sets of CDs appear together at a publisher in one of its subtrees. This offers publishers an added degree of protection against compromised routers inspired by k-anonymity techniques. r's summary, cd(r), is insufficient to pinpoint, even for the publishers in its subtrees, who advertises any given CD. Indeed, by ensuring that r's subtree contains sufficiently many publishers advertising sufficiently many distinct CDs as a protected group, we enable each publisher to remain anonymous by "hiding in the crowd" comprised of this group.

Third, under our assumptions, it is hard to impossible to get control over the edge routers (i.e., the leaf routers in a QDT) in order to probe the publishers directly for particular keywords or CDs based on the edge router's data summary. The key assumption is that each third party administrative domain controls a small fraction of nodes; therefore, it may control only a small number of edge routers. Moreover,

a single party does not have under its administration all routers of a given QDT. Even then, the data resides with the publishers and it gets released to the consumer only under the publishers' own access control policy.

Note that an edge router e cannot identify individual publishers updates. We do this by arranging publishers at edge routers participating in a particular QDT_i overlay to be protected in a k-anonymous group of other $k-1$ publishers. We may assume that edge routers do not keep record of advertised packets received directly from their connected publishers. Even if they do, we propose that publishers $P_1, .., P_k$ linked to e in QDT_i do not send individual index updates to e. Instead, we find the sum of the counts of the index of $P_1, .., P_k$, which is then sent out to router e, making e insensitive to which publisher in its k-group has published certain data. If e is compromised after publisher p registers with it, p's anonymity is preserved. Our scheme protects even against the case when p registers with an already compromised e. Indeed, as detailed in Section 2.4, the data summary implementation is hash-based and does distinguish among two distinct CDs with the same hash code. The publisher exploits this by only declaring the hash codes of its advertised CDs, instead of their actual value. Thereby, by compromising edge routers, an attacker may find out the association between set of CDs and its k-group of publishers, but not the exact one-to-one association mapping between a publisher and its advertised CDs.

Forth, it is obvious to see that compromising a publisher node p reveals all its private contents including advertised CDs $\mathsf{cd}(p)$ and local data store. At the other extreme, the best case scenario for breaking the system and for getting access to the publishers contents, is to compromise all nodes (e.g., by attacking the nodes) of a pure P2P network, such as the virtual scenario of a distributed *Facebook* network. However, this solution is infeasible in practice. Under our assumption of multi-administration domains, even if a small number of entities become untrusted (e.g., by direct attacks, by traffic analysis attacks, or by the Government's subpoena-ing one or a small number of ISPs) our design severely limits the amount of information gained about the network – not being able to collude information across different administration domains (e.g., ISPs).

Fifth, in general, we are interested to find out what is the cost to disclose A_2 if more than a node gets compromised and information from multiple compromised

nodes gets colluded together. In this context, an external attacker with the ability to compromise nodes, will take over node by node learning the actual connection of nodes on the QDT of interest (i.e., the children and parent links) as well as the local data summaries. The attacker moves on the parent-child links to explore the QDT. The process finishes when all seen data summaries report no further positive tests against the attacker's query.

Now, let us assume that compromising a node (regardless of its type: publisher, router) takes a certain cost c. It requires to compromise $N = fanout * log_{fanout}(n)/2$ nodes on average in order to disclose only one publisher from A_1, where n is the total number of nodes in the community network. Thus the cost is $c * N$. It requires $N = nr\ publishers * log_{fanout}(n)$ nodes to disclose A_1 fully, i.e., all publishers for a given CD. Note that N depends on how popular the CD is, the number of total nodes, the number of tree levels in the QDT overlay, and the fanout at nodes. In conclusion, the cost to disclose A_1 for a given CD is proportional to $O(logn)$. This result is a strong trust guarantee, depending on the application and its domain. In any case, we believe this is a good result since for centralized publishing systems and for DHT-based systems, it takes only one node to disclose A_1.

2.6 Experimental Setup

2.6.1 The Initial Overlay Network

To analyze the effects of our implementation choices on query dissemination, we built a simulator of a 10,000-node overlay network consisting of $N_p = 9,400$ publisher and $N_r = 600$ router nodes. The particular topology of this network is immaterial since in practice, every direct logical link between routers can be supported, and since actual dissemination will depend on the QDT overlays. We therefore assume a mesh topology allowing direct logical links for every pair of nodes.

2.6.2 A Real Data Set

To obtain true-to-life special-interest community, we simulate a distributed community that shares a real data collection, namely a partial XML dump of Wi-

kipedia, comprising about 1.1 million real Wikipedia documents which amount to a total size on disk of 8.6 GB [65]. that these documents are each brought into the community by one of the 9,400 publishers. Due to lack of information on which publisher generated which Wikipedia document, we assign Wikipedia documents to publishers based on a random uniform distribution. An interesting future work direction is to study how query dissemination is impacted if publishers generate clusters of semantically related documents.

2.6.3 CD Definition

Wikipedia, as well as other available XML sources such as CiteSeer, use structural schemata rather than ontological. This means that the majority of the tags on the root-to-leaf XML paths are concerned with the document organization, providing no semantic meaning. This observation motivates us to consider CDs defined as pairs (t, w), where w is a keyword and t gives the context in which w appears, given by the last XML element tag on the path from the root to w. We include this tag to support context-aware queries that go beyond standard keyword search. Moreover, we focus only on the tags that carry meaning to users. We restrict the last element tag to the following set: *"link", "b", "title", "subtitle" and "category"*. The combination of keywords and these contexts yields an interesting and complex set of about 3.2 million distinct CDs accounting for 24% of the set of all distinct CDs obtained by considering all possible tags[1].

2.6.4 Query Workload

We force the dissemination process to work under two extreme query types. We construct a family of 10 workloads $\{W_c^F\}_{1 \leq c \leq 10}$, each consisting of $5,000$ c-conjunct queries drawn at random from the space of queries with no match against the global collection. Similarly, we build the family of workloads $\{W_c^T\}_{1 \leq c \leq 10}$, each

[1]We can envision other CD definitions: including the entire path from the root to w to support more expressive queries, or keeping only w in support of only standard keyword search. We have tried all alternatives, obtaining analogous experimental results, which is why in the remainder of this thesis we report only the (last tag/keyword) case. The point we wish to emphasize is that the flexibility of CD definition is a key enabler for striking the right balance between expressivity of supported queries and space overhead.

comprising $5,000$ c-conjunct queries drawn at random from the space of queries with at least one match in the global collection. We also generate the $50,000$-query workloads $W^T = \bigcup_{c=1}^{10} W_c^T$ and $W^F = \bigcup_{c=1}^{10} W_c^F$. The matching query workloads increase the overall forwarding effort by forcing QDTs to send queries all the way to (some) leaves.

2.6.5 Scribe QDTs

Recall from Section 2.4 that, even in multiple-QDT configurations, the QDTs are isomorphic. We obtain a (unique up to isomorphism) QDT^S topology using Scribe [47]. We first convince ourselves of the faithfulness of the simulation, by generating a family of 20 Scribe tree topologies for the same node set (by varying the order in which the nodes join the network). We observe only non-essential variations across the family, thus boosting our confidence that picking any tree in this family is representative of Scribe's behavior. The particular Scribe tree we pick has $9,400$ leaf nodes and 600 internal nodes, 5 levels, average fanout of 16.7, and a maximum fanout of 101. The fanout features a very skewed distribution, decreasing from root to leaves (this holds for all 20 Scribe trees we considered). The distribution of the number of nodes per tree level is as follows: 1 node (the root) on the first level, 40 nodes on the second level (of which 3 are publishers), $1,189$ nodes on the third level, $6,163$ nodes on the fourth level and $2,607$ nodes on the fifth level. We determine the number k of isomorphic copies as in Section 2.4. We have $N_r = 600$ routers in total; among the 40 children of the root, 37 are routers. We obtain $N_u = 1 + 37 = 38$ and hence $k = \lfloor \frac{N_r}{N_u} \rfloor = \lfloor \frac{600}{38} \rfloor = 15$.

2.6.6 Fanout-balanced QDTs

For the sake of generality, we extend our simulation to QDT topologies not created by Scribe. We consider a topology QDT^B that uses the same router and publisher nodes, but eliminates the skewed fanout distribution that is typical of Scribe trees. This is beneficial since a node's fanout influences its forwarding cost. We first organize the 600 routers into a balanced skeleton tree with fanout 8, where levels $1, 2, 3, 4, 5$ have, respectively, $1, 8, 64, 512$ and the remaining 15 nodes. Next, we

connect the $9,400$ publishers to this skeleton tree, achieving for each node a fanout of 16 or 17. There are 75 non-leaf routers in the skeleton tree, and each receives 8 publishers, for a total fanout of 16. Among the leaf routers in the skeleton tree, 400 receive 17 publishers and 125 receive 16 publishers. We determine the number k of fanout-balanced QDTs in the usual manner: $k = \lfloor \frac{N_r}{N_u} \rfloor = \lfloor \frac{600}{9} \rfloor = 66$.

2.6.7 Metrics

Our goal is to improve the query throughput of the multi-QDT overlay, defined as the number of queries answered per time unit. Throughput is a manifestation of two more fundamental factors, namely the processing and forwarding effort at every router. For a given workload, the cumulative processing cost at node n is proportional to the number of query messages reaching n. The cumulative forwarding cost at n is proportional to the number of query messages it sends out to its children. The exact proportionality constant depends on factors we do not attempt to control, for instance such hardware properties as processor speed and network link bandwidth. We therefore design our metrics to separate out these factors and isolate the impact of our algorithmic solutions.

DEFINITION 2.6.1. (**Processing Load**) For a given workload W, we define the *processing load* at node n, denoted $PLoad_W(n)$, as the number of query messages reaching n across all QDTs it participates in. ◇

DEFINITION 2.6.2. (**Forwarding Load**) The *forwarding load* at n, $FLoad_W(n)$, is the number of query messages leaving n along all QDTs it participates in. ◇

Notice that none of the two measures is derivable from the other, since $FLoad_W(n)$ depends on n's fanout distribution (over the QDTs it participates in) and on the amount of pruning at n. For both load flavors, we define the *peak* load, which is the maximum load over all nodes. Clearly, decreasing either or both kinds of peak load results in increased throughput.

EXAMPLE 2.6.1. *In Example 2.2.1, 46 messages are used to disseminate 4 queries in 8 time units, while in Example 2.2.3, 50 messages disseminate the same query workload in 6 time units. Defining throughput as the number of queries answered*

*per time unit, the 4-*QDT *case has the higher throughput. The reason we don't simply use throughput as a metric is that it requires assumptions on the relative duration of processing and forwarding cost (in our running example, we take the simplifying assumption that forwarding cost takes constant time, independent of fanout).*

In Table 2.2, the processing load for a node is the number of queries on its row. For example, the processing load for node 13 is 4, which is also the peak processing load. In Table 2.4, the peak processing load is 3, experienced for instance by nodes 2 and 10.

*The forwarding load can be read by inspecting the transitions between columns and keeping track of parent-child relationships in the various trees. In the single-*QDT *case (Table 2.2), root node 1 has the highest peak forwarding load, 12 (it forwards each of the 4 queries to its 3 children). In the 4-*QDT *configuration (Table 2.4), the peak forwarding load is 6 messages, experienced by node 20 (1 message for Q_2, 3 for Q_3 and 2 for Q_4).*

Notice that, compared to the single QDT*, the 4-*QDT *configuration decreases both processing and forwarding peak load, which leads to improved throughput regardless of the concrete values of the per-query processing and forwarding cost.* ◇

The above considerations suggest comparing configurations by their degree of reduction of the peak processing and forwarding loads. Clearly, in any configuration, one cannot hope to lower the peak load below the average load, where *average processing load* is defined as

$$average\ PLoad = \frac{\sum_{n \in \text{routers}} PLoad_W(n)}{N_r}$$

and *average forwarding load* as

$$average\ FLoad = \frac{\sum_{n \in \text{routers}} FLoad_W(n)}{N_r}.$$

Notice that throughput, defined as number of queries answered in the time unit, is heavily proportional with the congestion effect, and in particular, with the peak load observed in the network, or the most loaded node. Therefore, ideal load balance is

achieved when the peak "drops" to the average load.

Note that our goal is not merely to achieve balance, as one can do so without improving throughput by simply raising the average load. Indeed, as discussed in Section 2.2, with increasing number k of QDTs both kinds of average load increase (though only slightly, as shown experimentally). This is because routing by smaller query blocks results in less pruning, which increases the overall number of messages. The smallest average loads are therefore witnessed in the single-QDT configuration, and they represent the ideal target for lowering the peak load. Since we are interested in closing the gap between the peak load in a k-QDT configuration and the ideal peak load, we report the *ideal-to-actual load ratio* metric as follows.

DEFINITION 2.6.3. (**Ideal-to-actual load reduction ratio**) In order to measure load balance-ness, we look at how close is the actual peak load in a k-QDT configuration from the ideal load. We define the *ideal-to-actual load ratio* to be the ratio between the peak load in the k-QDT and the average load in the single-QDT configuration or:

$$ideal-to-actual\ load\ ratio = \frac{peak\ load\ in\ the\ k-QDT\ configuration}{average\ load\ in\ the\ single-QDT\ configuration}$$

$\diamond$

2.7 Simulation Results

In this section, we explore through extensive simulations the space of configurations defined by the three dimensions given by the topology of QDTs, the number of QDTs, and the routing strategies. Our experiments confirm empirically that the configuration choices we advocate achieve near-optimal peak load reduction, and therefore near-optimal throughput.

2.7.1 Warm-up: Single-QDT Configuration

In this experiment, we confirm that the number of messages reaching the various levels in a single-QDT configuration is sufficiently skewed to justify our load-balancing efforts, in particular that the routers on the first two levels of the

tree bear the brunt of the load. For query workloads W_2^F and W_2^T, and the SCRIBE topology QDT^S, we report in Table 2.4 for every level the total and average number of messages seen by its nodes. Notice that the average number of messages per node decreases drastically below the upper two levels. Also notice that, unsurprisingly, workload W_2^T generates more overall messages, since its matching queries undergo less pruning than those in W_2^F.

Table 2.4: Messages per Level ($k = 1$, QDT^S, fully-informed)

QDT	W_2^F		W_2^T	
level	# msg. per level	Avg. # msg. per node	# msg per level	Avg. # msg. per node
1	5,000	5,000	5,000	5,000
2	200,000	5,000	200,000	5,000
3	173,066	146	636,507	535
4	28,509	5	513,464	83
5	4,869	2	193,575	74
Total	411,444	-	1,548,546	-

2.7.2 Effect of Number of QDTs

In this experiment, we validate our method for determining the number k of QDTs (recall Section 2.4). For workload W^T and fully-informed routing, we increase the number k of QDT^S copies from 1 to 31. Figure 2.5 shows the average and the peak load for both processing and forwarding.

Notice that with increasing k, the gap between the peak load and the average load decreases considerable. The highest load imbalance occurs for $k = 1$ as shown in the big gap between the peak and the average values for both the processing and the forwarding load. As predicted by our analysis in Section 2.4, $k = 15$ is indeed the "sweet spot" where the minimum gap is measured. Increasing k to 17 increases this gap. This is because the two additional cyclic permutations cause a "wrap-around" of the routers from the top two levels of QDT_1^S to the top two levels of QDT_{16}^S and QDT_{17}^S and thus introduce load imbalance. Also note that there is no point in looking at strict multiples of $\lfloor \frac{N_r}{N_u} \rfloor$ beyond $k = 15$, as they cost more overlay maintenance overhead without bringing the peak load any closer to the average load.

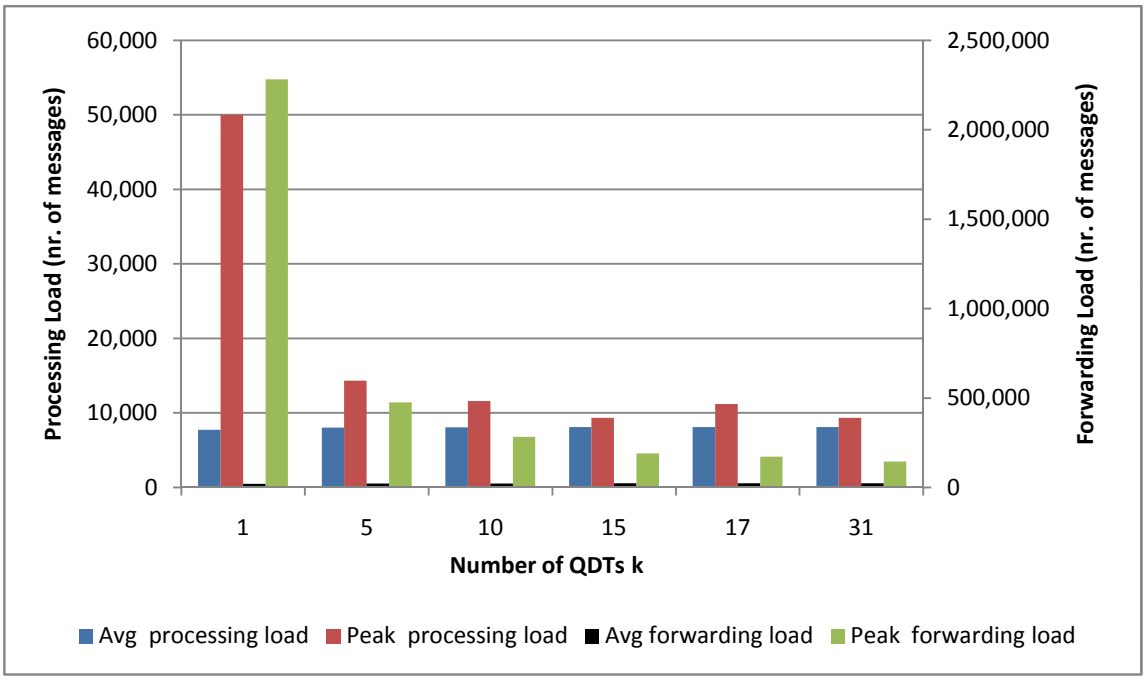


Figure 2.5: Effect of Number of QDTs (W^T, QDT^S, fully-informed routing, Processing and Forwarding load)

Finally, we observe that the negative effect of increasing overall number of messages with increasing k does occur: the average processing load is indeed the lowest for $k = 1$ since routing is done using all conjuncts, thus benefiting from maximum routing selectivity. However, the increase is very slow when compared to the decrease in peak load: the average processing load is $7,774$ for $k = 1$, $8,990$ for $k = 15$ and $9,039$ for $k = 31$, while the average forwarding load is $20,639$ for $k = 1$, $29,554$ for $k = 15$, and $29,998$ for $k = 31$. The negative effect of average load increase is outweighed by that of peak load reduction, as shown by the closing gap between peak and average loads.

We observe this behavior more accurately in terms of the ideal-to-actual peak load ratio, which for increasing k approaches the ideal value 1; therefore, closing the gap between the actual load and the ideal one. This can be seen in Figure 2.6, which reports the ratio between ideal and actual peak load ratio for both processing and forwarding loads. For example, the ideal-to-actual peak load ratio for processing load for the same values of k as in Figure 2.6 are, respectively: 6.43, 1.85, 1.49, 1.21, 1.44 and 1.20.

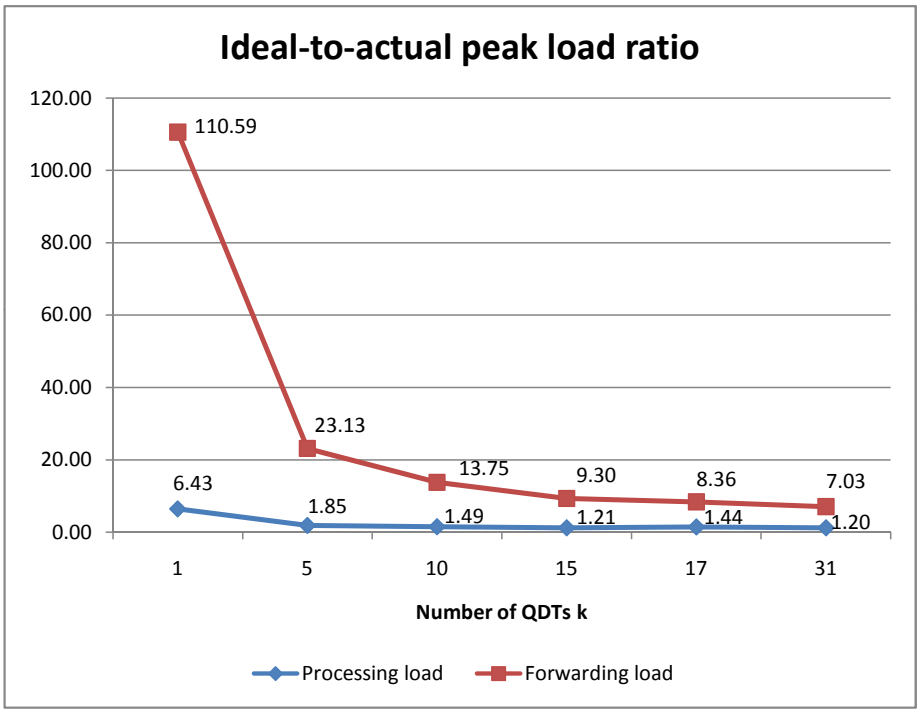


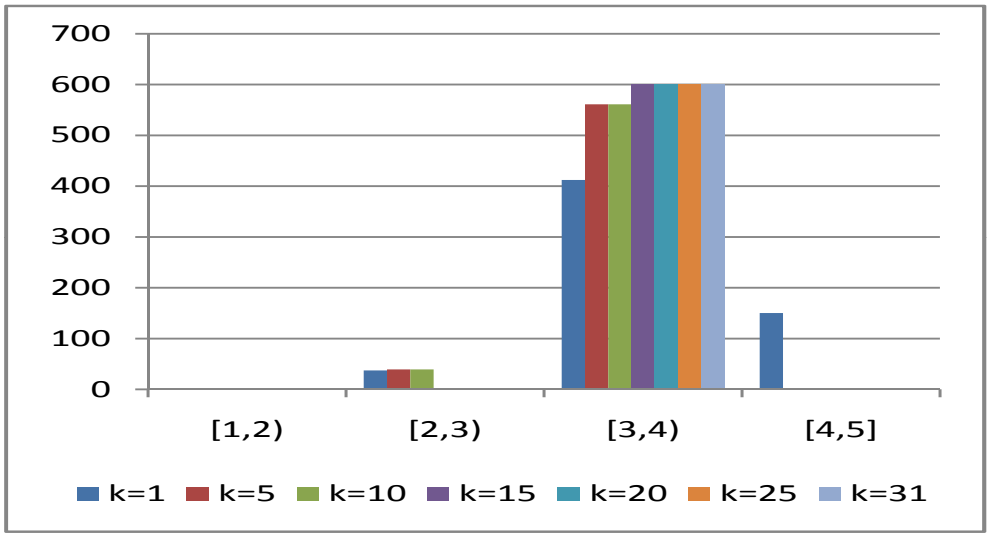
Figure 2.6: Effect of Number of QDTs (W^T, QDT^S, fully-informed routing, ideal-to-actual peak load ratio)

Figures 2.5 and 2.6 show the same trend for the forwarding load, with the only difference that, while the gap of peak and average loads decreases with growing $k \leq 15$ and saturates once k exceeds 15, we remain far from the ideal reduction (for which the ideal-to-actual load ratio is 1). This is explained by the forwarding load's correlation with the node fanouts and the fact that SCRIBE builds trees with highly skewed fanout distribution.

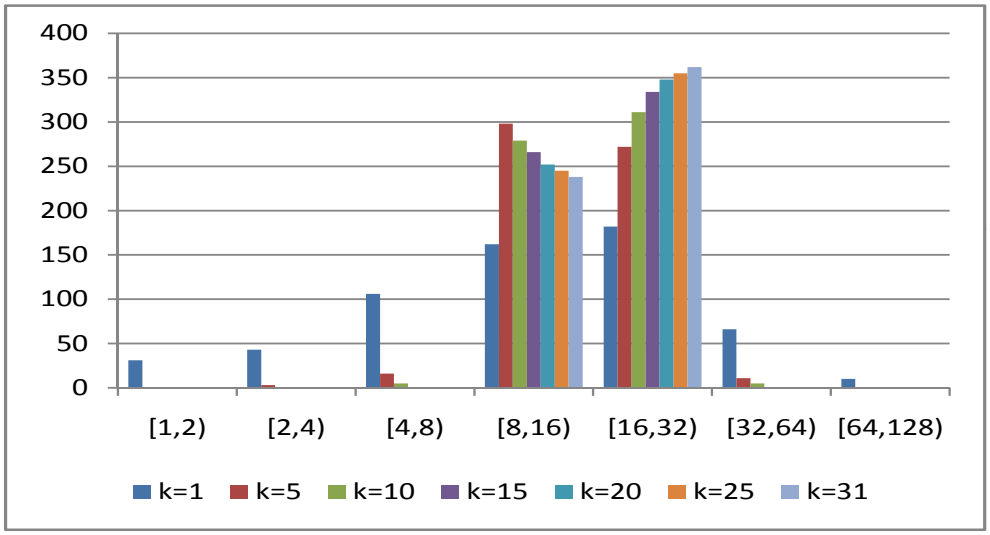
2.7.3 Effect of Static Load Indicators

We introduce two load indicators to capture statically the balance degree of a k-QDT configuration and confirm experimentally a good correlation with the dynamic query dissemination performance.

DEFINITION 2.7.1. (**Average tree level**) The *average tree level* for a node n, $ATL(n)$, is the average over all the levels of the k-QDT configuration node n participates in. ◇



(a) Average tree level (ATL) histogram.



(b) Average global fanout (AGF) histogram.

Figure 2.7: Distribution of internal nodes for ATL and AGF with the number of QDTs k (QDT^S)

Intuitively, the ATL distribution reflects the processing load distribution observed over all nodes at run time.

DEFINITION 2.7.2. (**Average global fanout**) Similarly, we define the *average global fanout* for node n, $AGF(n)$, as the average over all fanouts that n has when it participates in a k-QDT. ◇

Intuitively, the distribution of AGF values predicts the forwarding load distribution observed at run time. Figure 2.7 depicts the histograms for ATL and AGF as a variation of the number of QDT^{S}s from 1 to 31. This confirms that we get the best load balance with our techniques when ATL and AGF are fairly balanced. For example, when $k = 15$ the majority of the internal nodes are on the 3^{rd} or 4^{th} level in the UQDT on the average (Figure 2.7a), while the node fanout is distributed almost evenly into two intervals $[8, 16)$ and $[16, 32)$ on the average (Figure 2.7b).

Additionally, we can correlate the dynamic load behavior from Figure 2.5 with the spread of the static load indicators: the higher the ATL and the AGF spread, the higher the gap between the peak and the average loads (or, equivalently the ideal-to-actual peak load ratio) for the same k. Indeed, if $k = 1$, ATL presents the highest spread of values, showing an unbalanced distribution of nodes on tree levels. At the same time, Figure 2.5 shows the biggest gap between the peak and the average processing load. Just as the load gap decreases with the increase of k up to $k = 15$, the ATL distribution concentrates in the range of $[3, 4)$, which is nearly balanced. Similarly, we notice that an unbalanced distribution of node fanouts corresponding to $k = 1$ (which corresponds to a high AGF spread), correlates directly with the highest gap between the peak and the average forwarding load as in Figure 2.5. The gap closes in with k as the AGF spread becomes more balanced. However, notice that for $k = 15$ the fanouts are still not completely balanced. This explains why the ideal-to-actual peak load ratio doesn't quite reach 1. Our last experimental result in this section shows that balancing the fanouts is a key factor in bringing the peak forwarding load close to the average value.

2.7.4 Effect of Routing Strategy

We next compare the performance of the different routing optimizations as defined in Section 2.4 based on the query selectivity information. The idea is to use the routing state at a node to prune out routing choices, thus ruling out the low selective query blocks if possible. Among the query blocks that are left, we pick the most selective one based on a set of heuristics such as the number of conjuncts in the query block.

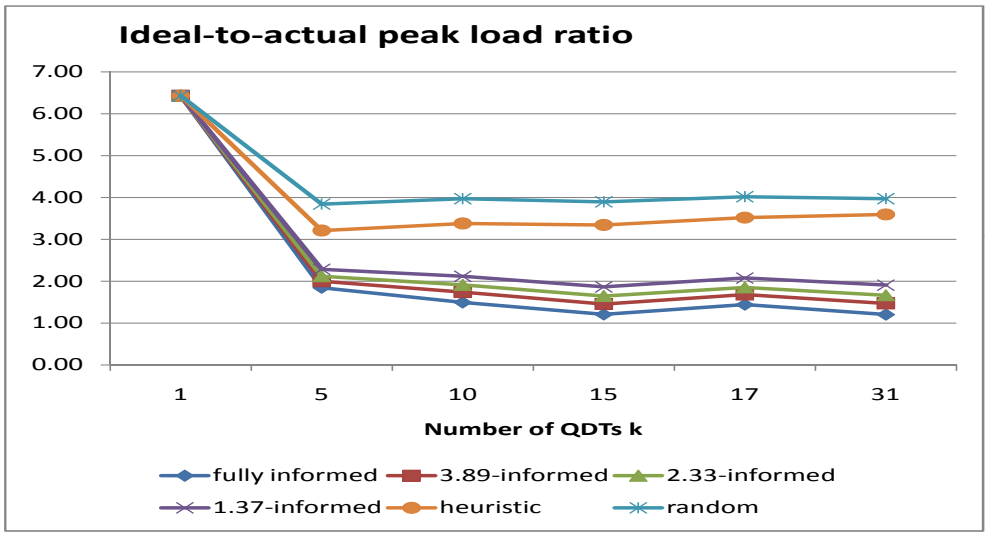
For the partially-informed strategy, we consider the case when publishers maintain the top s popular CDs for the following thresholds s = 43k, 74k and 124k, corresponding respectively to 1.37%, 2.33% and 3.89% of the total number of CDs in the global collection. We compare the strategies for workload W^T and QDT^S, reporting the ideal-to-actual peak load ratio in Figure 2.8, and the actual load behavior in Figure 2.9 and 2.10. We report the *routing selectivity benefit* as a qualitative measure of the peak load for partially-informed routings as distance from the peak load for the ideal routing strategy (i.e., fully informed).

First, we note that random routing performs worst, closely followed by heuristic routing. Both strategies are significantly outperformed by the (partially- or fully-)informed ones for every $k > 1$ (with the exception of $k = 1$ when all routing strategies coincide).

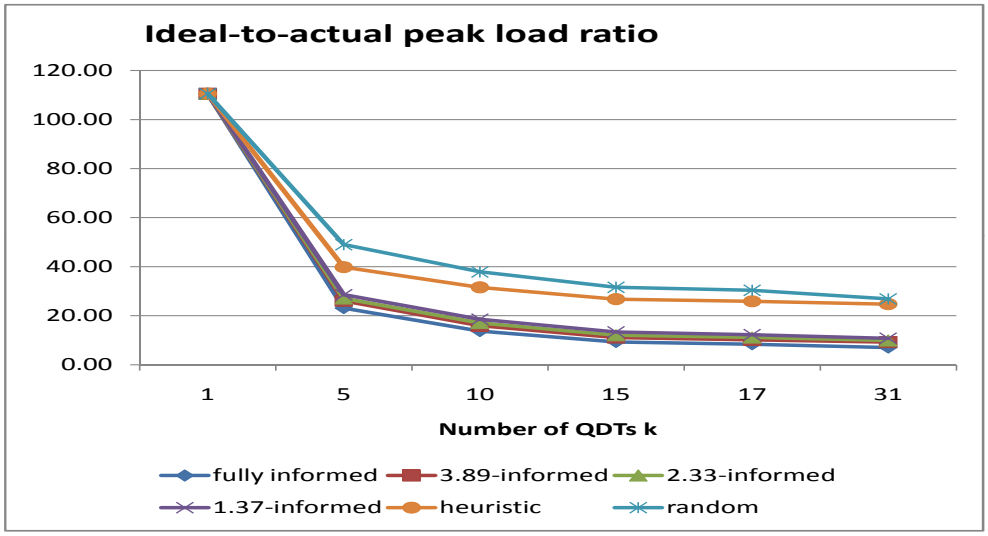
The family of informed routing strategies follows a common trend: with increasing $k \leq 15$, the gap between ideal and actual load shrinks drastically, reaches the sweet spot at $k = 15$ and essentially saturates for $k > 15$ (with a slight increase at $k = 17$ for processing load, due to the already discussed load imbalance introduced by the wrap-around).

Interestingly, random and heuristic routing behave slightly differently: at $k = 5$, they get closer to the ideal load than at $k = 15$. This behavior is caused by the following effect. The more QDTs, the more numerous the query blocks, which decreases the chance of a random pick hitting the most selective block. With increasing k, this effect starts generating non-minimal traffic, eventually canceling the load balancing effect. This explains why the random strategy degrades with increasing k. The reason the degradation saturates is that the number of query blocks cannot increase indefinitely (it must saturate once all blocks become singletons). Heuristic routing suffers from essentially the same problem: the more blocks we split a query into, the smaller the variation in block cardinality. Recall that, for same-cardinality query blocks, heuristic routing degenerates to random. In contrast, for the informed routing family, the experiments show that this effect remains subtle, being canceled out by the judicious choice of selective query blocks.

Finally, we observe that we can get very close to the benefits of fully-informed routing with negligible space overhead, by maintaining the frequency for even a small



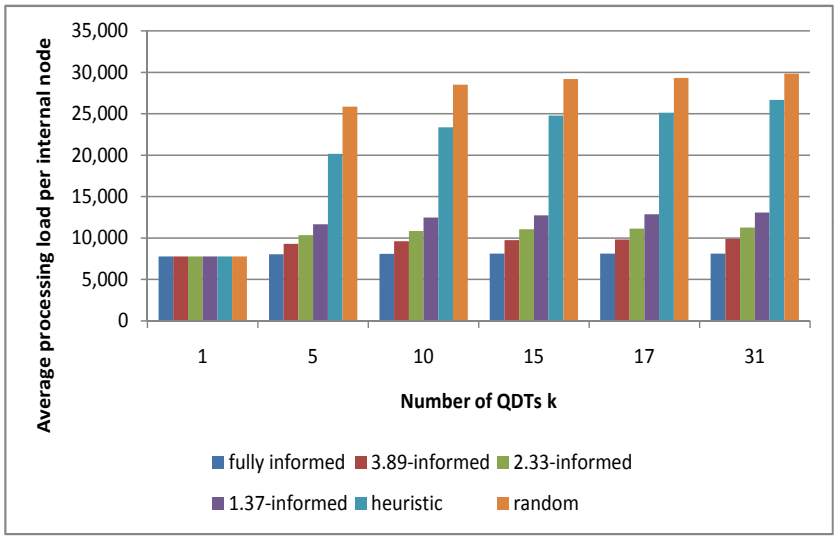
(a) Processing load.



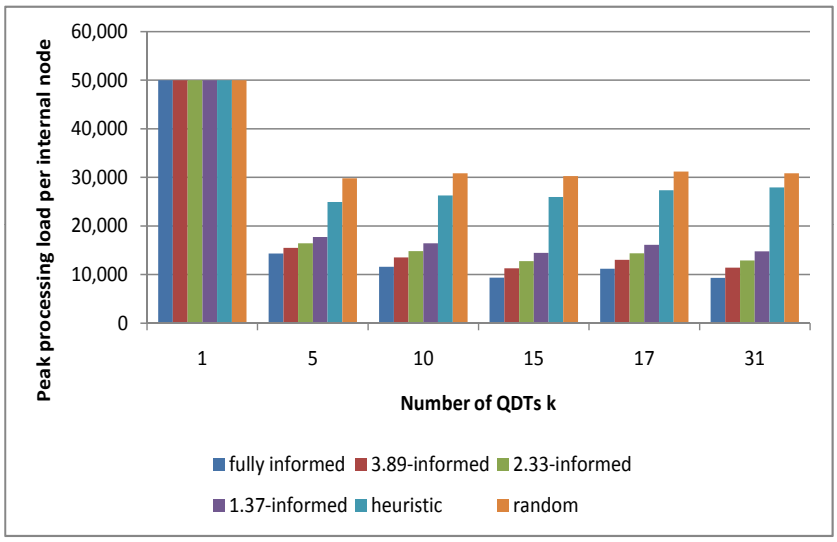
(b) Forwarding load.

Figure 2.8: Effect of Routing Strategy (W^T, QDT^S, ideal-to-actual peak load ratio)

fraction (e.g., 3.89%) of all CDs. We observe that the routing benefit varies linearly with the amount of maintained routing state. The more state is maintained, the higher the benefit of routing eliminating the redundant traffic; therefore, it improves the load balance. As a result, if we are given the amount of routing state, we can derive what is the amount of the routing benefit. For example, with only 3.89% state, we obtain 83% of the benefit according to Figure 2.8. These results strongly



(a) Average processing load



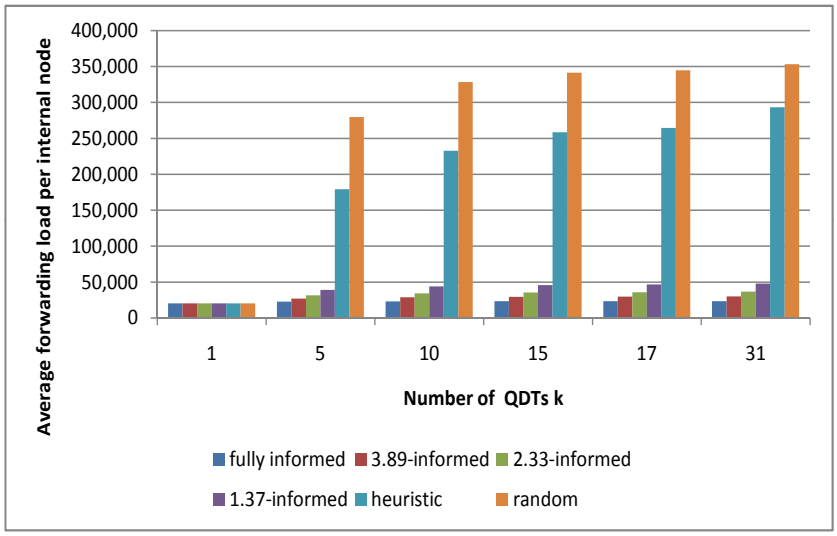
(b) Peak processing load

Figure 2.9: Effect of Routing Strategy (W^T, QDT^S, Processing load)

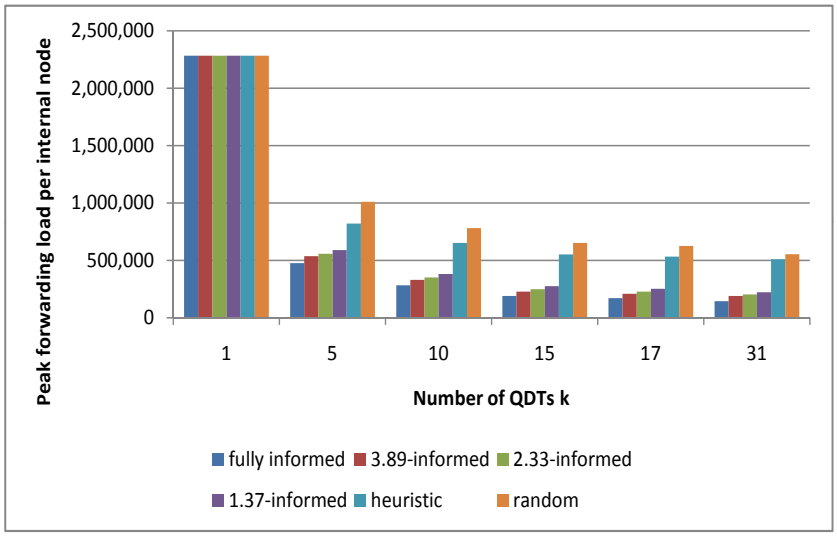
recommend partially-informed routing over the other strategies.

2.7.5 Effect of QDT Topology

We repeated all above experiments using the fanout-balanced QDT topology QDT^B, observing the same trends as for the SCRIBE topology QDT^S. We do not



(a) Average forwarding load



(b) Peak forwarding load

Figure 2.10: Effect of Routing Strategy (W^T, QDT^S, Forwarding load)

report the detailed results for lack of space. Instead, we summarize in Table 2.5 the comparison between the SCRIBE-generated and the fanout-balanced topology, relative to the peak load reduction. For query workload W^T and the fully-informed routing strategy, we show both the ideal and the actual value of the peak load ratio factor for the appropriate number of QDTs (15 for QDT^S and 66 for QDT^B).

Notice that both topologies come within reach of the ideal load reduction

Table 2.5: Effect of QDT Topology (W^T, fully-informed)

ideal-to-actual peak load ratio	SCRIBE (QDT^S) $k = 15$	fanout-balanced (QDT^B) $k = 66$
processing	1.21	1.18
forwarding	9.3	2.3

(when the ideal-to-actual load ratio is 1) for processing load. However, for forwarding load the SCRIBE topology misses the ideal by an order of magnitude, whereas the fanout-balanced topology only by a factor of 2.3. The main reason not even the QDT^B topology reaches the ideal forwarding load reduction is the inherent imbalance between the number of routers and publishers: the perfect configuration consists of a perfectly balanced tree whose internal nodes are routers and whose leaves are publishers. We did not simulate such a configuration because in practice we have no control over the numbers of routers and publishers.

Our experiments confirm that fanout-balanced topologies result in improved forwarding load reduction over SCRIBE topologies without sacrificing processing load reduction. As mentioned above, the benefit of using SCRIBE is of logistic nature, as it comes off-the-shelf with the overlay maintenance functionality. An advantage of our solution is its generality, in the sense that it assumes no control over the shape of the QDT, focusing on extracting the performance inherent in the topology.

2.7.6 Effect of Number of Conjuncts

It is a well-known fact that the more query conjunctions, the higher is the routing selectivity. To validate this conjecture on the UQDT infrastructure we increase the number c of conjuncts from 1 to 10 for W_c^T query workload. Figure 2.11 shows the behavior of the average load for one QDT^S and fully-informed routing. We observe a slow decrease in the processing load, while the forwarding load registers a quick decrease.

We present next the effect of number of conjuncts over the routing benefit using the various routing strategies introduced in Section 2.4. In addition to the previous conjecture, more conjunctions induce a larger number of query blocks. For

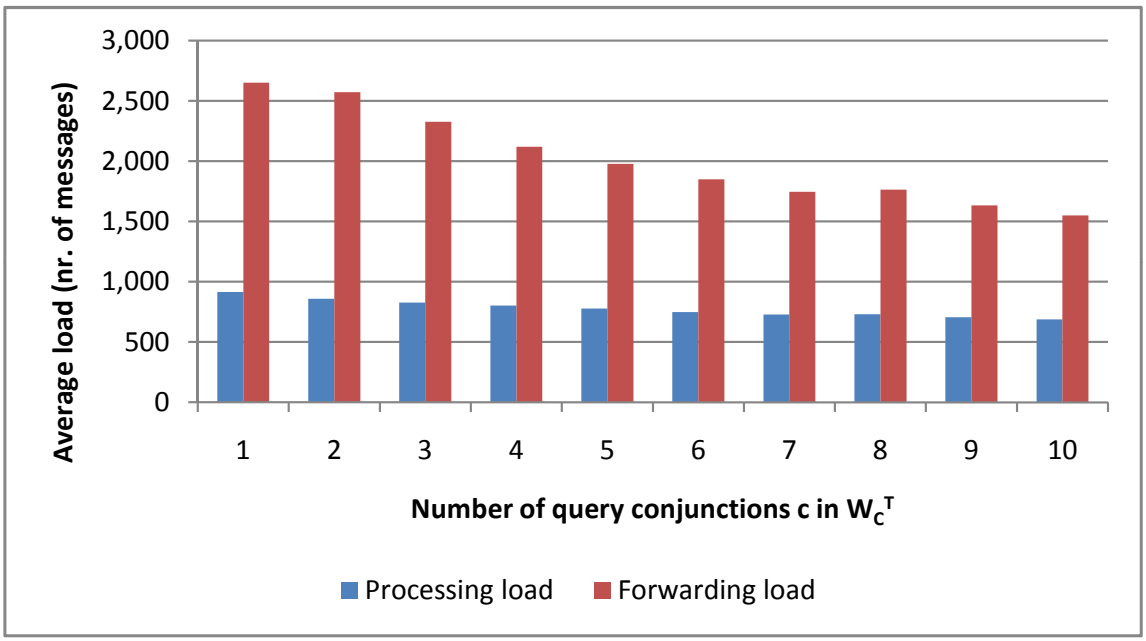


Figure 2.11: Effect of Number of Conjunctions (QDT^S, k=1, fully-informed routing)

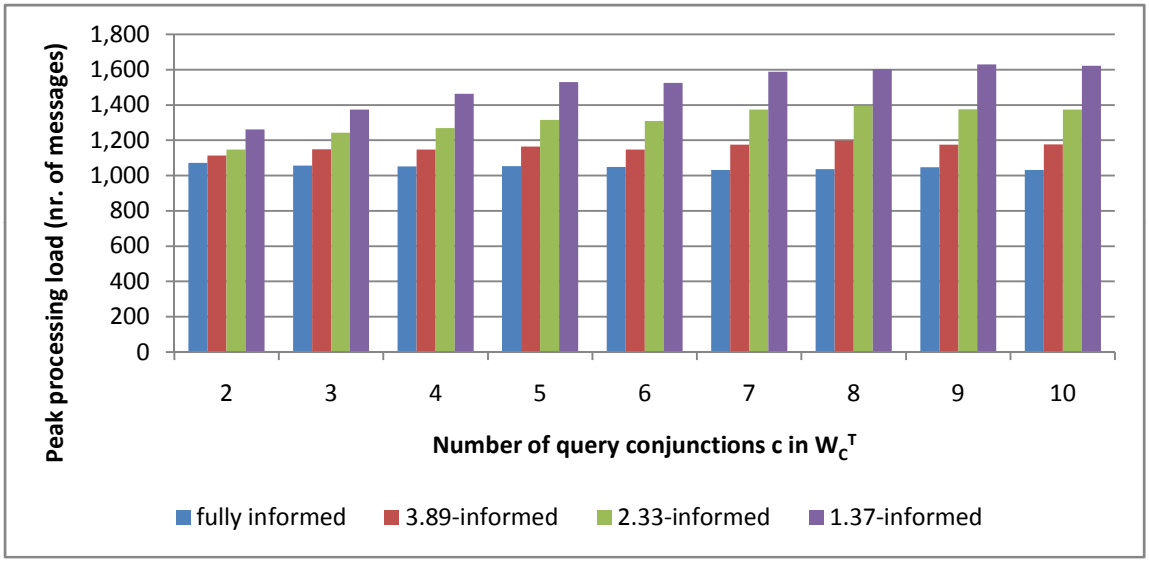
partially informed strategies, if none of these blocks fall in the set of popular maintained CDs at a node, then it means there are more query blocks to pick during query routing. Thus, the likelihood to randomly pick a high selective query block to route decreases, and therefore we expect the load to increase.

We vary both the routing strategy and the number of conjuncts c for W_c^T query workloads and k=15 QDT^S. Figure 2.12a shows the peak processing load behavior. We notice a slow increase in the gap between the peak load for fully informed routing and the peak for the other considered routing strategies. We measure the gap increase by reporting the routing benefit variation with the number of conjuncts for each of the partially informed routing strategies. Table 2.12b shows that the gap increase determines a decrease of up to $10 - 20\%$ of the routing benefit.

2.7.7 Latency Behavior

We consider a basic latency abstraction to be the number of hops that it takes to route a query workload from the root of the QDT to the publishers. Currently, we ignore queuing delays during query routing.

We observe that the latency varies very little with the number of QDTs. It



(a) Processing load.

	3.89-informed	2.33-informed	1.37-informed
W_2^T	0.96	0.93	0.85
W_3^T	0.92	0.85	0.77
W_4^T	0.92	0.83	0.72
W_5^T	0.90	0.80	0.69
W_6^T	0.91	0.80	0.69
W_7^T	0.88	0.75	0.65
W_8^T	0.86	0.74	0.65
W_9^T	0.89	0.76	0.64
W_{10}^T	0.88	0.75	0.63
W^T	0.82	0.73	0.65

(b) Routing benefit for processing load.

Figure 2.12: Effect of Number of Conjunctions (QDT^S, k=15)

is strongly influenced by the dissemination's tree topology being puled toward the the tree levels with the most populated leaves. Since in a QDT^S most of the leaves carrying data are on level four, the average latency varies a little between 3.5 and 3.88.

The small latency variation can be interpreted as a result of loosing the accuracy of routing selectivity: the more QDTs, the higher the likelihood of touching false positive nodes at deeper levels than the true positive answers.

2.8 Related Work

We identify three research directions related to our work. To provide high throughput and scalable search over distributed content there have been proposed mediation based, replication based and partitioning based solutions. We analyze each of them in terms of efficiency, query power expressivity and resistance to censorship.

Mediation approach. In the mediator approach the data residing with different publishers in the network is collected and accessed via a single site, also called the mediator. This architecture is the standard for most of the current search engines and online hosted communities. Often, a mediator requires a highly parallel backend in order to scale and to remove congestion. Yet, the presence of only one central point of access to data is vulnerable to attacks and moreover, it compromises the preservation of a censorship resistant environment. In addition, a mediator approach restricts the business model of the publishers as they are disintermediated from the consumers.

To leverage the already existing computational power of the network, recently there has been a large body of work that focuses on finding only the peers with relevant data to a user's query. These methods construct data summaries at nodes and use them as routing indices [60] to disseminate the query in the network toward the relevant publishers. Hybrid approaches such as [57, 90] compromise the decentralization premise by utilizing super-peer nodes to coordinate the storage and data retrieval. We look next at complete decentralized architectures.

Replication based approaches. One way to increase data availability and to balance the load, and therefore to improve the system throughput is to replicate all or parts of the data (or indices of it) redundantly at the router nodes [94, 77]. Disseminating queries to publishers in such a scenario is simple since each such router has global information. Nevertheless, maintaining multiple redundant routers incurs increased cost in store space and in updates to the index. Moreover, information can be exposed easier to an attacker since all the routers require now protection as opposed to protecting only one site as in the mediator approach.

Partitioning based approaches. As a result, a better way to leverage the distributed computational power is based on data partitioning. The baseline for this

approach is to partition the data items such that a partition block is managed by different peers. The impact on query processing is that conjunctive queries span over multiple blocks, therefore resolving them means less processing selectivity. The advantage of the approach is twofold. First, there is no index update overhead and no extra space requirement compared to the mediator solution. Second, the publishers and the consumers do not need to know the details of the partitioning scheme to send data or queries. The network takes care of identifying the relevant matching data to the queries. Our approach is partition based.

DHT based. A partition-based solution to building routing indices that is popular among structured P2P networks is to leverage distributed hash tables (DHTs). A DHT provides a distributed logical abstraction of object identifier lookups (e.g. mostly filename lookups) over the physical underlay. However, this approach focuses more on handling atomic queries [77, 104, 40, 127, 22, 38] and less on the efficiency of complex queries processing [82, 119]. Our UQDT solution is optimized for Boolean (XPath) queries. Another body of work builds hierarchies of overlays based on DHTs. To improve locality, the hierarchies are created based on the document content similarities [129] or on the nodes proximity in the network to minimize latency [126, 97, 30, 78].

However, DHTs are inappropriate for the problem we study, since DHT nodes maintain *complete* knowledge of all the publishers that advertise specific data items. An attacker can gain global information for data items by simply compromising a single DHT node. In contrast, no UQDT node maintains complete knowledge about any data item. In addition, UQDT nodes only maintain summaries which are masked union of data items present in their subtrees, without knowing exactly which data item is contained in which publisher. A leaf QDT router node only has information about a small set of publishers that connect to it. An attacker needs to compromise a significantly large number of UQDT nodes before gaining any global information.

Other routing strategies. [88, 59] is a class of structured P2P indices that is not DHT-based. Their focus is on the design of efficient tree topologies for distributed dissemination.

Koloniari and Pitoura [91] consider the problem of routing path queries over

schema-less XML documents in a P2P system. They propose the use of one hierarchical overlay network that clusters nodes with similar XML documents where nodes contain filters that summarize repositories of a set of its neighbors to facilitate path query routing. This approach is similar to the single-QDT configuration, with the attendant limitations, and our technique for maximizing throughput has the potential to be useful for this problem as well. At the opposite spectrum, [71] builds a QDT for each published data item which can sometimes be impractical due to the large number of CDs.

P2P publish/subscribe. A complementary problem is that of distributed publish/subscribe, wherein query subscriptions from users are maintained in a distributed index structure, and data items are disseminated to subscribers as soon as they are published. Topic-based approaches use a set of pre-defined static topics to form the rendezvous points between subscribers and publishers, and some form of multicast is often used for efficiency purposes, such as IP multicast [37], generic application-level multicast [43], and multicast on top of DHTs [47]. Although constructed for a different goal than QDTs, we show how off-the-self SCRIBE trees [47] for data dissemination can be used for QDTs as well.

Content-based publish/subscribe approaches match the entire published content against (possibly aggregated) subscriptions. A good example of this approach is ONYX, a system for XML content dissemination [66], wherein a dissemination tree is rooted at each publisher. Each router maintains for each interface an aggregate subscription (XML query) that summarizes all the subscriptions downstream along that interface. A published data item starts from the root (the publisher), and gets forwarded to all downstream interfaces whose corresponding aggregate subscriptions match the data item. Chand and Felber [48] take a similar approach. SemCast [100] aggregates subscriptions in a centralized manner using a cost-based model, and documents are routed through the network based on the subscription aggregates.

Censorship resistant. Most of the existing censorship-resistant systems like Eternity [29], Free Haven [67], Publius [124], Tangler [123] are based on anonymizing the communication, and therefore anonymizing the end-to-end communicating entities (Tor [68], Freenet [54], Freedom [42], Tarzan [74], MorphMix [103]). This is usually done by using proxy based services (e.g., Anonymizer.com, JAP [10]), based

on DHTs which comes with their disadvantages, or based on trusted servers to encrypt and route the traffic through established anonymous tunnels over the other nodes. Instead, our work is necessary and it complements the censorship resistant work with the DLA mechanism. Our DLA preserves privacy of users in publishing communities in the sense that it provides anonymity for the association of users with certain information.

2.9 Conclusions

The dawn of the age of online communities poses the challenge of empowering information publishers to join democratic communities and query their global data collection in an ad-hoc fashion. We present an infrastructure that meets this challenge by allowing data to reside with its owners and by supporting queries against the global data collection, with no need for any central authority that disintermediates publishers from consumers. These queries are evaluated by dissemination to relevant publishers under censorship resistant constraints, using a distributed index structure. Since the dissemination indices are subject to potential attacks and censorship, our solution precludes third parties from learning the exact publisher/CD associations without compromising a significant fraction of the router nodes.

Technically, our approach is dual to the conventional work on data dissemination, and its viability depends on the feasibility of efficient query dissemination. Our contributions towards proving feasibility range from identifying the design space (with its trade-off dimensions, relevant metrics and notion of optimality), to introducing solutions that achieve near-optimality with only low overhead.

Partially-informed routing emerges as the best-value strategy, with low space overhead to yield the same benefits as fully-informed routing, and to significantly outperform random and heuristic routing. The solution exploits crucially the dual role of QDTs, deploying them as both query and statistics dissemination trees. While we show that fanout-balanced topologies are closest to optimal, an advantage of our solution is its generality, in the sense that it focuses on extracting the performance inherent in any given topology. This enables the seamless porting of our techniques on top of off-the-shelf overlay maintenance tools developed by networking research.

In future work, we contemplate extensions of our solution in two stages. The first is to incorporate ranking functions and exploit the UQDT overlay for top-K query processing. The second stage targets the support of expressive queries that allow complex filtering conditions on the CD matches. These conditions pertain to both the keywords and the context they appear in (a representative of this class of queries is W3C's XQuery Full-Text extension [114]). This line of work will exploit our framework's generality with respect to the definition of CDs.

2.10 Acknowledgements

Chapter 2 is currently being prepared for submission for publication of the material, which is joint work with Alin Deutsch, K.K. Ramakrishnan, and Divesh Srivastava. The dissertation author was the primary investigator and author of the paper.

Chapter 3

Evaluation of XML Full-Text Queries at Publishers

3.1 Need for Rich Text Search on Semi-structured Data

The ability to search both the structure and text content of XML documents is gaining importance with the increase of large XML repositories such as the United States Library of Congress (LOC) documents [11], medical data in XML such as HL7 [7], and the IEEE INEX [9] (TREC [18]-like effort but for XML) data collection [9]. Querying XML repositories rich in text content requires sophisticated full-text search features ranging from matching individual keywords to combining matches with Boolean operators, proximity distance (including keyword-distance, -window, and -order) and number-of-times-to-repeat conditions, stemming, and stop words.

XML querying is a well-studied topic, with several powerful database-style query languages such as XPath 2.0 [112] and XQuery 1.0 [113] set to become W3C standards. The XQuery expression given below is a typical query on the US Library of Congress repository that selects congressional bills with actions that relate to "non-immigrant status", such as bills that amend the Immigration and Nationality Act. The query returns the descriptions of such bills that have been introduced since 2002:

```
for $b in //bill
where fn:contains($b//action, "non-immigrant status")
  and $b//action-year >= 2002
return
  <bill> {$b/description} </bill>
```

The query applies the XQuery substring-matching function *fn:contains* [115] to the text nodes contained in `action` elements. As discussed in [24], sub-string functions in XQuery cannot express more complex full-text queries, such as restricting the order and distance between words. These limitations are due to the XQuery data model, which does not represent positions of words in input documents. Word positions are necessary to compute distance and to evaluate order predicates. Therefore, even if custom XQuery functions were defined for each full-text search primitive, they would not be fully composable without extending the data model.

XQuery Full-Text [114] is an extension of XQuery that supports fully composable full-text search primitives defined on a data model of words and positions. The language is inspired by TeXQuery [24], a proposal to the W3C Full-Text Task Force. XQuery Full-Text provides powerful full-text search primitives such as simple word search, Boolean queries, word distance as well as stemming, regular expressions and stop words. XQuery Full-Text also supports scoring and top-k ranking of query results. We refer to the XQuery Full-Text search primitives as *FTSelections*. All *FTSelections* are defined on a data model, called *AllMatches*, which represents words and their positions in documents. Because the semantics of each *FTSelection* is defined in terms of operators on the *AllMatches* data model, the *FTSelections* are fully composable.

The key problems when implementing XQuery Full-Text are : (i) choosing a representation for the *AllMatches* data model; (ii) implementing the semantics of each full-text primitive on *AllMatches*; and (iii) processing input documents to provide the word positions used in *AllMatches*. In GalaTex,[1] our strategy is to employ XML and XQuery directly to solve these problems. First, we implement the *AllMatches* data model in XML itself [114]. Second, we implement each full-text primitive as a native XQuery function that takes one or more *AllMatches* values

[1] *http://www.galaxquery.org/galatex*

and produces an *AllMatches* value. Last, we pre-process each input document to produce auxiliary XML documents that map each word to their positions in the input documents; these auxiliary documents are accessed by the semantic functions. This implementation strategy is both general and expedient. By using XML and XQuery themselves to implement XQuery Full-Text, we were able to rapidly prototype a complete implementation of the language. In addition, our technique can be used with any XQuery implementation.[2] In particular, we describe in this chapter an implementation on top of GALAX, a complete XQuery implementation.

XQuery Full-Text supports scoring and ranking of query results and permits any ranking method that satisfies the XQuery Full-Text scoring requirements [114, 116]. In GALATEX, we adapt the probabilistic relational algebra [76, 105] to *AllMatches* by extending each full-text primitive with the ability to manipulate scores. Our implementation satisfies the XQuery Full-Text scoring requirements.

When implementing GALATEX, we have focused more on completeness and conformance than on efficiency. By focusing on completeness, GALATEX can serve as a reference implementation of XQuery Full-Text and as a platform for experimenting with new research ideas for scoring XML data, optimizing XML queries on both structure and content, and evaluating top-k queries. Ultimately, we want GALATEX to be both complete *and* efficient. One of GALATEX's performance bottlenecks is the size of the *AllMatches* values generated by each *FTSelection*. We discuss several ways of optimizing the evaluation of *FTSelection*s, including logical rewritings of the full-text query and the optimization of XML queries on both structure and content.

In particular, this chapter makes the following contributions:

- We present a general technique for implementing XQuery Full-Text using an existing XQuery implementation.

- We describe GALATEX, the first complete implementation of XQuery Full-Text, a W3C specification that extends XPath 2.0 and XQuery 1.0 with full-text search predicates. GALATEXis an all-XQuery implementation (i.e., implemented almost entirely in XQuery itself) initially focused on completeness and conformance. In addition to a command-line interface, GALATEX includes a

[2]See *http://www.w3.org/XML/Query for a list of XQuery implementations.*

browser interface that permits users to execute both the XQuery Full-Text use cases [117] and their own queries.

- We adapt the scoring method for the probabilistic relational algebra [76, 105] to *AllMatches* and show that this adaptation satisfies XQuery Full-Text's scoring requirements.

- We identify some performance challenges, possible solutions, and their interactions with existing XQuery implementations.

We begin with an overview of XQuery Full-Text in Section 3.2. Section 3.3 describes general implementation techniques and their realization in GalaTex. Advanced evaluation strategies are considered in Section 3.4. We conclude and present the related work in Section 3.5.

3.2 Preliminaries: XQuery Full-Text Language

We introduce XQuery Full-Text (XQFT) search and scoring through examples and highlight some key features of the language. We refer the reader to the language specification [114] and the language use cases [117] for more details on the language.

3.2.1 Full-Text Search

XQuery Full-Text extends XQuery with a full-text search expression (*FTContainsExpr*) and with a scoring function (*ft:score()*). The *FTContainsExpr* takes an evaluation context (i.e., a sequence of XML nodes) and a full-text search (*FTSelection*) condition and returns a Boolean value that is true if and only if some node in the evaluation context satisfies the condition. Because *FTContainsExpr* is a first-class XQuery expression, full-text search is seamlessly integrated into XQuery and XPath. In particular, since *FTContainsExpr* returns a value in the XQuery data model (i.e., a Boolean value), it can occur wherever a Boolean value is permitted in other XQuery expressions. The following expression illustrates the interaction of full-text search with an XPath expression.

```
//book[.//section ftcontains "usability" && "testing"]/title
```

The expression returns the titles of books with at least one section that contains the search tokens *usability* and *testing*. The *FTContainsExpr* is used as a predicate that returns a Boolean value. Its evaluation context is an XQuery expression, i.e., *.//section* within *//book*, and its *FTSelection* is *"usability" && "testing"*. This query also illustrates how XQuery Full-Text uses existing XQuery constructs such as path expressions to specify the evaluation context and the returned nodes (*/title*).

An *FTSelection* may be used to express matching individual words (*FTWord*), Boolean connectives between keywords (*FTAnd*, *FTOr* and *FTNegation*), order predicates (*FTOrdered*), proximity distance between words (*FTDistance* and *FTWindow*), scoping within sentences and paragraphs (*FTScope*) and the ability to specify the number of occurrences of words (*FTTimes*). The query below illustrates how these primitives can be combined. It returns true if some book in the evaluation context (*//book*) contains the tokens *usability* and *testing* in the same sentence within a window of five words.

```
//book ftcontains "usability" && "testing" same sentence window 5
```

XQuery Full-Text can also embed XQuery expressions. The expression below returns true if some article in the evaluation context contains an occurrence of a title of one of Paul Auster's books. The XQuery expression *//book[./author = "Paul Auster"]/title* specifies the search tokens, and the keyword *any* specifies that at least one of the titles can occur in the articles.

```
//article ftcontains (//book[./author = "Paul Auster"]/title) any
```

In addition to *FTSelection*s, XQuery Full-Text has a rich set of matching modifiers called, *FTMatchOptions*, such as stemming, stop-words, regular expressions, case sensitivity, diacritics, special characters, synonyms, languages, and ignoring specified XML subtrees [26]. *FTMatchOptions* operate at the level of individual words and can be seamlessly composed with any *FTSelection* to modify how the full-text search is performed. The expression below returns true if some book in the

evaluation context contains any tokens derived from *usability* and *testing* after applying stemming. For example, a book that contains *user* and *tests* would satisfy the full-text search condition because *usability* and *user* share the stem *use*, and *testing* and *tests* share the stem *test*.

```
//book ftcontains "usability" && "testing" with stemming
```

The last example below is similar to the one above, but requires that search tokens occur within a window of five words, ignoring stop-words when computing this window.

```
//book ftcontains "usability" && "testing"
                  with stemming window 5 without stopwords
```

3.2.2 Full-Text Scoring

The previous expressions all yield Boolean values, but often users require the results of full-text search to be scored and ranked by the quality of the match. In XQuery Full-Text, scoring is achieved using the second-order function *ft:score()*, which returns one score for each node in the set of input XML nodes. This function is second order because it accepts an *FTSelection* expression, not a value, as an argument – it is also the only second-order function in XQuery.

The score of a node captures its relevance to an *FTSelection*. For example, the expression below returns a sequence of scores for each book in the evaluation context.

```
let $scores := ft:score(//book,
    "usability" weight 0.8 && "testing" weight 0.2)
```

Note that user-specified weights can be applied to compute score. In this example, *usability* is given a weight of 0.8 and *testing*, a weight of 0.2. The exact means by which *ft:score* uses these weights is implementation-defined.

The *ft:score()* function provides the framework for supporting different scoring mechanisms, but does not dictate the exact scoring mechanism itself. This flexibility is necessary, because vendors are unlikely to agree on the same scoring technique. In

fact, scoring for XML is an active area of research (e.g., see [56, 75, 80, 83, 98, 118]), and many vendors view scoring techniques as product differentiators. However, there are two properties that every scoring mechanism must satisfy [116]: (i) the score of a node in the evaluation context must be 0 if and only if the node does not satisfy the full-text condition specified in *FTSelectionWithWeights*. Otherwise, its score must be in the interval (0,1]; (ii) for the nodes in the evaluation context, a higher score value implies a higher degree of relevance to *FTSelectionWithWeights*.

The *ft:score()* function returns a sequence of floating-point numbers, which may occur wherever a number is permitted in other XQuery expressions. This enables the expression of powerful queries such as the one below, which computes the top-10 results for the previous query.

```
for $result at $rank in
 (for $node in //book
  let $score := ft:score($node,
     "usability" weight 0.8 && "testing" weight 0.2)
  order by $score descending
  return <result score="{$score}">{$node} </result>)
where $rank <= 10
return {$result}
```

The inner FLWOR expression returns the results in descending order by score, and the outer FLWOR expression only returns the top ten of these results.

Our last example illustrates how *FTContainsExpr* and *ft:score()* can be combined to search based on one condition and score based on another one. The expression below selects books that contain *usability* and *analysis*, and these books are scored based on *usability* and *testing*.

```
for $book in //book[. ftcontains "usability" && "analysis"]
let $score := ft:score($book, "usability" weight 0.8 &&
                              "testing" weight 0.2)
return <result score="{$score}"> {$book} </result>
```

3.3 XQuery Full-Text Implementation

Numerous strategies exist for implementing XQuery Full-Text – as many strategies as there are for implementing XQuery itself! Possible strategies include extending an existing XQuery engine with native support for the XQuery Full-Text data model and operators; extending an existing full-text search engine to serve as an XQuery Full-Text co-processor; or translating XQuery and XQuery Full-Text into another query language, such as SQL. XQuery Full-Text relies on the *AllMatches* data model that captures words and their positions. Regardless of the implementation strategy chosen, the key implementation problems are representing the *AllMatches* data model, implementing the semantics for each *FTSelection*, and making the word positions used in the input documents accessible to the *AllMatches* data model.

Because new languages benefit from the rapid development of experimental implementations, our strategy was to employ XML and XQuery directly to implement XQuery Full-Text. We first describe key implementation techniques and then their realization in GalaTex.

3.3.1 General Implementation Techniques

Preprocess Documents & Queries

In the XQuery data model, the text node is the smallest unit representing document content, but in the XQuery Full-Text data model, the smallest unit is a word and its position within a document or phrase. We define an XML value, called *TokenInfo*, to represent a word and its position in an input document or in a search phrase. Two preprocessing steps yield *TokenInfo* values: the text in input documents is tokenized off-line, and the search phrases in a full-text query are tokenized at query evaluation time.

A *TokenInfo* value contains a word and a unique identifier that captures the relative position of the word in a document or in a phrase. When tokenizing document text, a *TokenInfo* may also contain the XML node, sentence, and paragraph that directly contain the word. The DTD for a *TokenInfo* value is below.

```
<!ELEMENT TokenInfo (Token, Identifier, Node?, Sentence?, Para?)>
```

As an example, Figure 3.1 contains a tokenized document in which each word in the text has a corresponding *TokenInfo* identifier, which contains the global position of the word in the document. This information could be augmented with the appropriate node, sentence and paragraph identifiers.

We define abstract functions for pre-processing search phrases and documents. Tokenization of a search phrase is performed by the *getSearchTokenInfo()* function, which takes a search string and returns a sequence of *TokenInfo*s. We explain in Section 3.3.1 how the match-options argument is used during tokenization.

```
getSearchTokenInfo($searchPhrase as xs:string,
   $matchOptions as FTMatchOptions) as TokenInfo*
```

The following abstract functions access tokens and their positions in documents. The *getTokenInfo()* function takes an evaluation context of zero or more element nodes and a search word specified as a *TokenInfo* value and returns all the positions of the word in the given evaluation context. The *getPositions()* function is similar, but restricts the evaluation context to one element node. *getTokenInfo()* and *getPositions()* can both be defined in terms of the *containsPos()* function, which returns true if the given evaluation context contains the given word. The *wordDistance()* function returns the distance between two words given any match options that might affect the *FTWindow* or *FTDistance* primitives.

```
getTokenInfo($evalContext as element()*,
             $searchToken as TokenInfo ) as TokenInfo*
getPositions($node as element(),
             $searchToken as xs:string ) as TokenInfo*
containsPos($node as element()*,
            $searchToken as TokenInfo ) as xs:boolean
wordDistance($token1 as TokenInfo,
             $token2 as TokenInfo,
             $mo as FTMatchOptions ) as xs:integer
```

```
<book(1)>
  <author(2)>Millicent(3) Marigold(4)</author>
  <content(5)>
    <p(6)> The(7) usability(8) of(9) software(10) measures(11) how(12) well(13) the(14)
           software(15) provides(16) support(17) for(18) quickly(19) achieving(20)
           specified(21) goals(22).
    </p>
    <p(23)> The(24) users(25) must(26) be(27) and(28) feel(29) well-served(30).
           Software(31) usability(32) is(33) a(34) good(35) measure(36) of(37) that(38).
    </p>
  </content>
  <title(39)>Conquering(40) the(41) systems(42)</title>
</book>
```

Figure 3.1: XML document fragment with positions

The AllMatches Data Model

An *AllMatches* value specifies all possible position solutions to a full-text search query and can be viewed as a propositional logic formula in disjunctive normal form (DNF) [24]. We represent instances of the *AllMatches* data model using XML values that conform to the following DTD:

```
<!ELEMENT AllMatches (Match)*>
<!ELEMENT Match (StringInclude|StringExclude)*>
<!ELEMENT StringInclude TokenInfo>
<!ELEMENT StringExclude TokenInfo>
```

Each *Match* in an *AllMatches* corresponds to one of the disjuncts in the DNF formula. Each *StringInclude* in a *Match* corresponds to the proposition that the evaluation context node must contain a word position, and each *StringExclude* specifies that the evaluation context node should not contain a word position.

The FTSelections

Each *FTSelection* function takes one or more *AllMatches* values and returns one *AllMatches*. For example, consider the sample document in Figure 3.1 annotated with word positions and the following full-text query, which returns those books that contain paragraphs containing words similar to *usability* and *software* case sensitive within ten words of each other:

```
//book[.//p ftcontains ("usability" with stemming) &&
  ("software" case sensitive) with distance at most 10 words]/title
```

Each full-text query has an associated query evaluation plan of *FTSelections*. Figure 3.2 contains the plan for the above query. We distinguish between two stages in the evaluation plan. The bottom two levels of the plan construct *AllMatches* values using the position functions described in Section 3.3.1. Once we have the first *AllMatches*, all the other primitives manipulate *AllMatches* only.

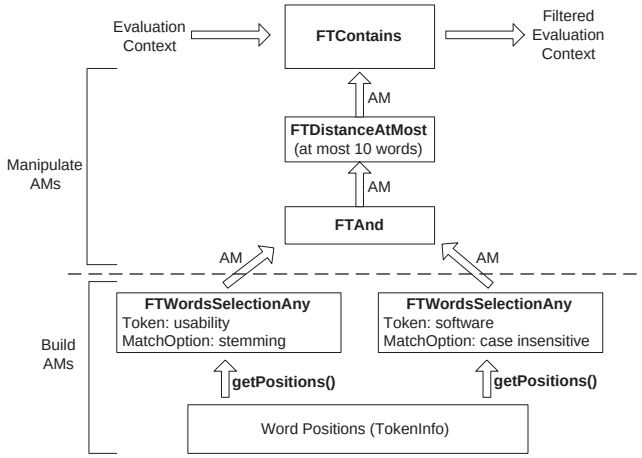


Figure 3.2: Full-Text XQuery evaluation plan

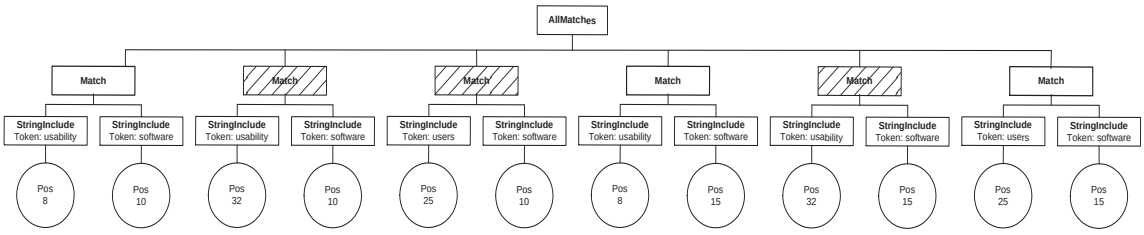


Figure 3.3: *AllMatches* for "usability" with stemming && "software" case sensitive

A variety of primitives build *AllMatches* depending on the query search criteria: a single word (*FTSingleSearchToken*), any word or all words from the set of given phrases (*FTWordsSelectionAnyWord*, *FTWordsSelectionAllWord*), and any or all phrases (*FTWordsSelectionPhrase*, *FTWordsSelectionAny*, *FTWordsSelectionAll*).

In Figure 3.2, the two *AllMatches* representing the tokens *usability* and *software* become the inputs to the *FTAnd* primitive. The resulting *AllMatches* is given

in Figure 3.3. It contains six possible *Matches*. These *AllMatches* are further filtered by the *FTDistance* primitive. The final *AllMatches* contains only the first, fourth, and sixth *Matches* (see Figure 3.3). Once those matches are generated, they are passed to *FTContains()*, the top-most node in Figure 3.2, in order to filter the XML nodes in the evaluation context.

The Match Options

Match options are modifiers that apply to each of the search words. If a match option is not specified explicitly in the query, then its default value is used. The default match options are: *case insensitive, without special characters, without regular expressions, without stemming, without stop words, element content is not ignored, English-language selected, without thesaurus, and diacritics insensitive* [114]. When a match option is specified explicitly in the query, it overrides the default for the phrases to which it applies. The XML representation of match options is:

```
<!ELEMENT FTMatchOptions (FTMatchOption)*>
<!ELEMENT FTMatchOption (FTCaseOption | FTDiacriticsOption |
  FTSpecialCharOption | FTThesaurusOption|FTStemOption |
  FTRegexOption | FTLanguageOption | FTStopWordOption
  | FTIgnoreOption)>
```

The abstract function *applyMatchOption()* applies all match options from *FTMatchOptions* to a list of search tokens and returns the token information for all the modified search words.

```
applyMatchOption($mo as FTMatchOptions,
                 $searchToken as xs:string* ) as TokenInfo*
```

3.3.2 GalaTex Implementation

We describe how these general techniques are realized in our GalaTex architecture depicted in Figure 3.4. A demonstration of GalaTex and of the XQuery Full-text use cases are available at: *http://www.galaxquery.org/galatex/*. GalaTex is implemented on top of the GalaxXQuery engine [72],[3] a complete XQuery im-

[3] *http://www.galaxquery.org*

plementation that supports functions and modules.

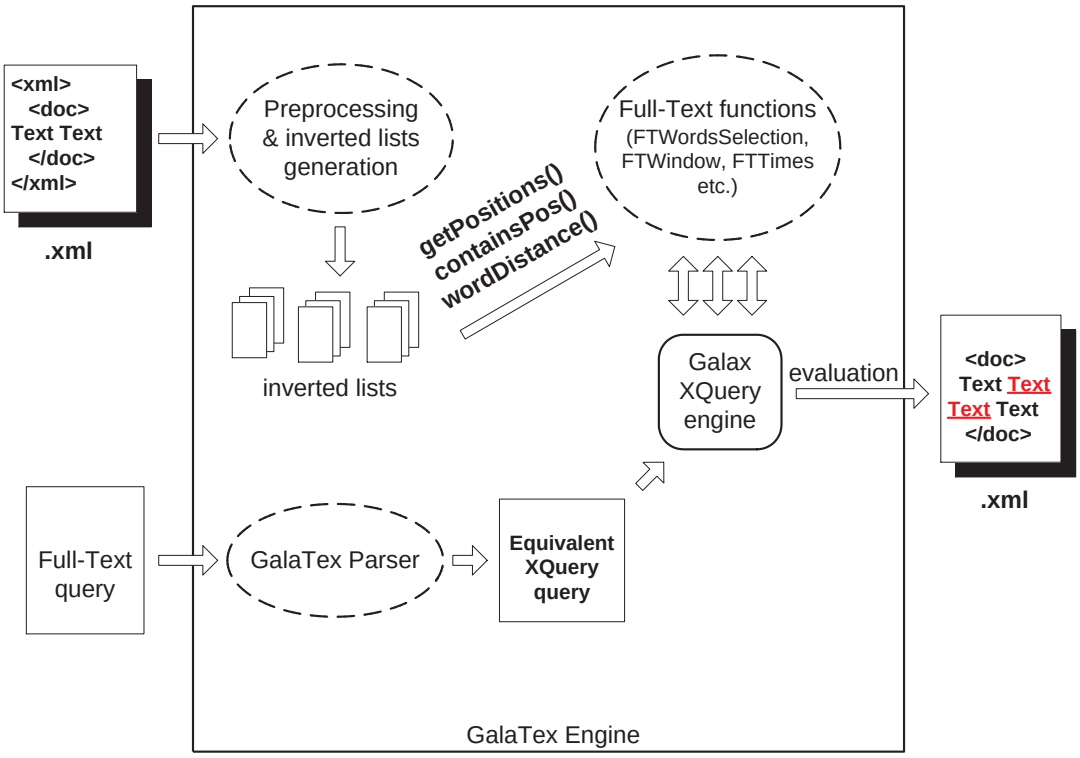


Figure 3.4: Architecture of GalaTex

In the upper left of Figure 3.4, GalaTex preprocesses input documents, and for each distinct word, produces one document containing all the positions of that word, represented by *TokenInfo* values. These documents essentially contain *inverted lists*, which map words to their positions. These inverted-list documents are the inputs to *getPositions()* and related functions.

In the lower left of Figure 3.4, GalaTex translates XQuery Full-Text queries into equivalent XQuery queries by mapping each *FTSelection* into a call to the corresponding XQuery function. The XQuery functions themselves (upper right of Figure 3.4) are implemented in an XQuery library module, where each function implements one *FTSelection* primitive.

These functions use the *getPositions* and related functions to access token positions in the inverted-list documents and generate an *AllMatches* value. These *AllMatches* are then composed by the full-text functions and the final *AllMatches* is used to filter the specified context nodes of the input query.

On the right of Figure 3.4, GALAXtakes the input documents, the translated query, and the library module of XQuery functions, evaluates the translated query, and yields the result as an XML document. The final result contains the relevant XML document fragment in which the search words are highlighted.

The GALATEX library module uses XQuery's optional schema import and validation features, which are supported by GALAX. These features are not required by our implementation, but are useful because they guarantee that all *AllMatches* and *FTMatchOptions* values are valid instances of the corresponding types.

Document Preprocessing

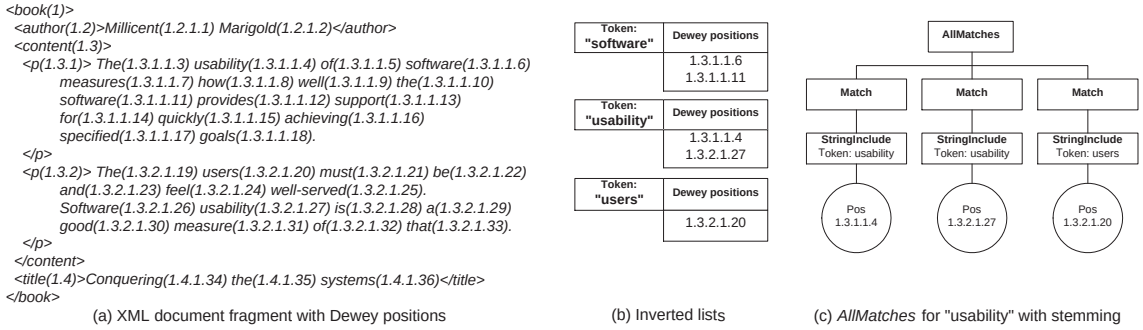


Figure 3.5: Dewey positions and AllMatches example

The document pre-processing step is done off-line and the result is a set of documents that contain *TokenInfo* values. Our tokenizer assumes that words are delimited by punctuation and whitespace symbols as in English. We chose to implement the *TokenInfo* identifier using Dewey numbering [110]. The Dewey number encodes the depth-first node path from the document root to each node. For each word, the identifier contains the Dewey number of the node containing the word appended with the word's absolute position in the document. For example, in Figure 3.5(a), the first occurrence of *usability* has identifier 1.3.1.1.4, indicating it is contained in the node with identifier 1.3.1.1 and it is the fourth word in the entire document. For each distinct word identified during tokenization, we create one inverted-list document that contains all of the word's *TokenInfo* values. Figure 3.5(b) contains the inverted lists for *software*, *usability*, and *users*.

We chose to represent the inverted lists in XML format. The benefit is that

all the abstract functions that manipulate positions described in Section 3.3.1 are expressed as XQuery functions operating over XML values. For example, the XQuery implementation of *getTokenInfo()* is given below.

```
declare function
  fts:getTokenInfo( $evalCtx as element()*,
                    $searchToken as fts:TokenInfo)
  as fts:TokenInfo*
{
  for $node in $evalCtx,
      $pos in fts:getPositions($node, $searchToken/@word)
  return
    <fts:TokenInfo word="{$searchToken/@word}"
       prefixPos="{fn:string($pos/@prefixPos)}"
       absPos="{fn:string($pos/@absPos)}"/>
}
```

For each node `$node` in the evaluation context, and for each occurrence of the search word in that node, a *TokenInfo* value is returned. The *getPositions* function accesses the inverted list for `$searchToken` and returns only those positions that are included in `$node`. Testing whether a word position is contained XML node is done in *containsPos()* operator, which compares the integer components of Dewey values hierarchically (e.g., 1.10.1 > 1.9.2). Although the Dewey representation for positions is a character string (i.e. "1.3.1.43") comparing two Dewey positions is not just a simple string comparison. The comparison function, *CompareHierarchicalStringsByValue()*, has to take into account the decimal representation of each component of the Dewey encoding.

Query Parsing & Translation

As shown in Figure 3.4, the GalaTex parser translates a full-text query into an equivalent XQuery query. This design was chosen to improve portability, to avoid direct impact on the GalaxXQuery engine, and to speed implementation.

This design also allowed us to implement and test subsets of the XQuery Full-Text specification quite easily while treating the XQuery engine itself as a black box.

Currently, the parser replaces each full-text expression in the original query with the appropriate composition of *FTSelection* function calls. Match options are propagated to the relevant *FTWordsSelection* calls. For example, consider the following full-text query:

```
//book[.//p ftcontains ("usability" with stemming) &&
  ("software" case sensitive) without stemming with
  distance at most 10 words ordered]/title
```

The GalaTex parser produces the following XQuery query:

```
//book[
  ( let $ec_1 := (.//p ) return
  fts:FTContains( $ec_1,
    fts:FTOrdered(
      fts:FTWordDistance(-1, 10,
        fts:FTAnd(
          fts:FTWordsSelectionAny( $ec_1, "usability",
            fts:MO_FTStemOption("with stemming",
            <fts:FTMatchOptions/>), "1"),
          fts:FTWordsSelectionAny( $ec_1, "software",
            fts:MO_FTStemOption("without stemming",
            fts:MO_FTCaseOption("case sensitive",
            <fts:FTMatchOptions/>)), "2"))))))
]/title
```

In the translated query, each search string has been replaced with a call to the *FTWordsSelectionAny* function. In addition to the search string, each of these calls also passes the evaluation context (i.e., *//book//p*), the applicable match options, and the position of the search string in the original query. Although it is used in multiple calls, the evaluation context is bound to a variable and is only evaluated once. Note also that the match option *without stemming* has been propagated into

each *FTWordsSelectionAny* call (except for *usability* with the explicit *with stemming* override).

The operators '*&&*', '*distance*' and '*ordered*' have been replaced with calls to the full-text functions *FTAnd*, *FTWordDistance* and *FTOrdered*, respectively. The *FTOrdered* function uses the position information for each search string in the query to ensure that words are considered in the order in which they appear in the query.

To do this translation, the parser requires and uses very little knowledge of the XQuery language. In fact, only three tokens were added to our grammar to handle the XQuery language and its overlap with the XQuery Full-Text grammar. These additional tokens:

1. identify the start of XQuery expressions and sub-expressions in order to extract the evaluation context for a full-text expression,
2. identify the return to XQuery from a full-text expression, and
3. disambiguate between parenthesized XQuery expressions and parenthesized full-text expressions in order to identify XQuery expressions embedded within a full-text expression.

Since XQuery code can contain full-text expressions which, in turn, can contain XQuery expressions, arbitrary nesting of the languages is possible and is supported by the parser. In the following example, the result of the embedded XQuery expression is used as a search string.

```
//book[.//p ftcontains
    (//book[./author ftcontains "Marigold"]/title)
    with stemming window at most 15]/title
```

The translated XQuery for the above expression is:

```
//book[
  ( let $ec_1 := ( .//p ) return
    fts:FTContains( $ec_1,
      fts:FTWindow(-1, 15,
```

```
        fts:FTWordsSelectionAny( $ec_1,
          (//book[
            ( let $ec_2 := ( ./author ) return
              fts:FTContains( $ec_2,
                fts:FTWordsSelectionAny( $ec_2,
                  "Marigold",
                  <fts:FTMatchOptions/>, "1")))
          ] /title),
          fts:MO_FTStemOption( "with stemming",
            <fts:FTMatchOptions/>), "2"))))
]/title
```

As expected, all of the XQuery-specific code is passed unchanged to the XQuery engine, and each full-text expression has been replaced with an *FTContains* function call (even in the case where one full-text expression is nested inside another). Note that each embedded XQuery expression in the original query must be enclosed in parentheses.

Query Evaluation

Manipulating the Positions We chose to represent the inverted lists in XML format. The benefit is that the operators that manipulate positions are expressed as XQuery functions operating over XML values. For example, the expression for *getTokenInfo()* is given below.

```
declare function
  fts:getTokenInfo( $evalCtx as element()*,
                    $searchToken as fts:TokenInfo)
  as fts:TokenInfo*
{
  for $node in $evalCtx
  let $positions:= fts:getPositions($node, $searchToken/@word)
  where not(empty($positions))
  return
```

```
    for $pos in $positions
    return
      <fts:TokenInfo word="{$searchToken/@word}"
      prefixPos="{fn:string($pos/@prefixPos)}"
      absPos="{fn:string($pos/@absPos)}"/>
}
```

Intuitively, it returns for each node `$node` in the evaluation context all the occurrences of the search word `$searchToken` by the means of *getPositions()* function. The latter function checks in the file corresponding to the inverted list of the given searched word and returns only those positions that are included in the given node. Testing whether a position is inside an XML node is done in *containsPos()* operator by simply testing the Dewey positions. Although the Dewey representation for positions is a character string (i.e. "1.3.1.43") comparing two Dewey positions is not just a simple string comparison. The comparison function, *CompareHierarchicalStringsByValue()*, has to take into account the decimal representation of each component of the Dewey encoding (i.e. ”1.10” > ”1.9”). This is the XQuery expression for *getPositions()*:

```
declare function
  fts:getPositions( $node as element(), $searchToken as xs:string )
  as element(token)*
{
  if (glx:file-exists(fn:concat($searchToken,".xml")))
  then
  for $entry in
    fn:doc(fn:concat($searchToken,".xml"))/invlist
      /entry[fts:containsPos( $node,
      <fts:TokenInfo word="{$searchToken}"
      prefixPos="{@prefixPos}" absPos="{@absPos}"/>)]
  return
    <token>
    {
```

```
        attribute absPos    { $entry/@absPos },
        attribute prefixPos { $entry/@prefixPos },
        attribute name      { $searchToken }
      }
      </token>
  else ()
}
```

As seen above, the position extraction is done on a per node basis. We would like to improve it by plugging in the *getTokenInfo()* operator a sort-merge implementation of the two input lists: the nodes in the evaluation context and the nodes in the inverted lists. The inverted lists should be organized as a list of XML node ids each having associated a list of positions. This way the node id will not be repeated for each word position as in Figure 3.5. For example, the inverted lists of a given word should contain the most specific XML nodes that contain it together with a list of positions where that word occurs in this node. The inverted lists are ordered by node id.

This can be done by keeping the same implementation and replacing the functions that manipulate positions with external functions. This is necessary because the sort-merge algorithm requires an imperative implementation whereas XQuery is a declarative language. Thus one step in the evolution of the implementation would be to implement these functions natively in an imperative language. Moreover this is possible because XQuery supports external functions. This approach works also for XQuery Full-Text implementations that are using native indices other than inverted lists.

FTSelections A library module of XQuery functions implements the semantics of the *FTSelection* primitives. We return to a simplified version of the query in Section 3.3.1 to illustrate how these functions work.

```
//book[.//p ftcontains ("usability" with stemming) &&
  ("software" case sensitive) with distance at most 10 words]/title
```

The equivalent XQuery expression generated by the parser is:

```
//book[
( let $ec_1:= ( .//p ) return
  fts:FTContains( $ec_1,
    fts:FTWordDistance(-1, 10,
      fts:FTAnd(
        fts:FTWordsSelectionAny( $ec_1, "usability",
          fts:MO_FTStemOption( "with stemming",
          <fts:FTMatchOptions/>), "1"),
        fts:FTWordsSelectionAny( $ec_1, "software",
          fts:MO_FTCaseOption( "case sensitive",
          <fts:FTMatchOptions/>), "2")))))
]/title
```

This translation corresponds to the query plan in Figure 3.2. We describe this plan "bottom up", beginning with the inner-most function calls to *fts:FTWords-SelectionAny* and ending with the outer-most call to *fts:FTContains*. Note that this code is valid, executable XQuery code and not merely a pseudo-code description of a query plan.

The first function, *fts:FTWordsSelectionAny*, constructs the initial *AllMatches*. It calls the *FTSingleSearchToken()* function whose definition in XQuery expression is below.

```
declare function
  fts:FTSingleSearchToken(
     $evalCtx as element()*,
     $searchToken as fts:TokenInfo,
     $matchOptions as fts:FTMatchOptions,
     $queryPos as xs:string ) as fts:AllMatches
{
  <fts:AllMatches>
  {
    for $position in fts:getTokenInfo($evalCtx, $searchToken)
    return
```

```
        <fts:Match>
          <fts:StringInclude
            queryString="{$searchToken/@word}"
            queryPos="{$queryPos}">{ $position }
          </fts:StringInclude>
        </fts:Match>
    }
    </fts:AllMatches>
  }
```

The above function obtains the positions of the search token and constructs one *AllMatches* that contains one *Match* per position. This function uses *getTokenInfo()* described in Section 3.3.2. We defer discussion of match options to Section 3.3.2. The last argument to *FTSingleSearchToken()* is (*$queryPos*), which is a variable that contains the relative position of the search word in the full-text query. It is used in conjunction with *FTOrder*. Figure 3.5(c) shows the *AllMatches* for *usability with stemming*.

The *AllMatches* values constructed for *usability* and *software* are inputs to the *FTAnd* function, which computes the Cartesian product of their *Match*es as follows:

```
declare function
  fts:FTAnd( $allMatches1 as fts:AllMatches,
             $allMatches2 as fts:AllMatches)
  as fts:AllMatches
{
    <fts:AllMatches>
    {
      for $match1 in $allMatches1/fts:Match,
          $match2 in $allMatches2/fts:Match
      return
        <fts:Match>
        { $match1/*, $match2/* }
        </fts:Match>
```

```
    }
    </fts:AllMatches>
}
```

This function computes all possible pairs of *Matches* for *usability* and *software* and returns an *AllMatches* value. This value is input to *FTDistance*, which selects those matches that satisfy the distance condition as follows:

```
declare function
  fts:FTWordDistanceAtMost(
    $n as xs:integer,
    $allMatches as fts:AllMatches,
    $matchOptions as fts:FTMatchOptions)
  as fts:AllMatches
{
  <fts:AllMatches>
  {
    for $match in $allMatches/fts:Match
    if fn:empty($match/fts:StringInclude) then
     $match
    else
     let $sorted:= for $si in $match/fts:StringInclude
                   order by $si/fts:TokenInfo/@absPos
                   ascending return $si
     where every $idx in (1 to fn:count($sorted) - 1)
           satisfies fts:wordDistance(
                         $sorted[$idx]/fts:TokenInfo,
                         $sorted[$idx+1]/fts:TokenInfo,
                         $matchOptions) <= $n
     return
       <fts:Match>
         { $match/fts:StringInclude }
         {
```

```
          let $sortedStrMatch:=
                       for $si in $match
                       order by $si/*/fts:TokenInfo/@absPos
                       ascending return $si
          for $stringExcl in
            $sortedStrMatch/fts:StringExclude
          where some $stringIncl in
                $sortedStrMatch/fts:StringInclude
                satisfies fts:wordDistance(
                             $stringIncl/fts:TokenInfo,
                             $stringExcl/fts:TokenInfo,
                             $matchOptions) <= $n
          return $stringExcl
         }
       </fts:Match>
  }
  </fts:AllMatches>
}
```

Intuitively, the matches that satisfy *FTDistance* are those for which each pair of adjacent positions satisfy the distance condition. For each of these matches, the included positions and only the excluded positions that fall in the specified distance range are returned.

Finally, *FTContains* filters the evaluation context and returns only those nodes that contain at least one match that satisfies all the inclusion and exclusion constraints.

```
declare function
  fts:FTContains( $evalCtx as element()*,
                  $allMatches as fts:AllMatches)
  as xs:boolean {
   some $node in $evalCtx
   satisfies
```

```
     (some $match in $allMatches/fts:Match
      satisfies fts:satisfiesMatch($node, $match))
};

declare function
  fts:satisfiesMatch( $node as element(),
                      $match as fts:Match )
  as xs:boolean {
  ( every $stringInclude in $match/fts:StringInclude
    satisfies fts:containsPos($node,
                $stringInclude/fts:TokenInfo))
  and
  ( every $stringExclude in $match/fts:StringExclude
    satisfies not(fts:containsPos($node,
                $stringExclude/fts:TokenInfo)))
};
```

Match Options In GalaTex, match options are translated by the parser into a set of match option functions implemented in XQuery. A match option has the effect of expanding one search word to a set of words that becomes the new set of search words for the current full-text query. This expansion occurs in *FTSingleSearchToken()*, described in Section 3.3.2, which applies *applyMatchOption()* before calling *getTokenInfo()* for the current search word. The function returns the positions of all words that result from applying match options to the current search word. This interface is quite flexible and it allows plugging in any match option implementation.

To implement match options we used the XQuery Functions and Operators defined in [115], in particular, *fn:matches*, *fn:replace*, *fn:lower-case*, *fn:upper-case*. Hence, any XQuery implementation that supports them can be used with GalaTex.

For example, to find the expansion set of a search word when the *case* match option is set to *case insensitive*, we compare for equality the search word with each distinct word from the input document. The list of distinct words is generated in

the preprocessing step. Both words are filtered by *fn:lower-case* function as in:

```
let $sToken:= $searchTokens/@word
for $docToken in fn:doc("list_distinct_words.xml")/
  ListDistinctWords/invlist/@word
return
  if (fn:lower-case($docToken)=fn:lower-case($sToken))
  then $docToken
  else ()
```

The same technique works when the *regular expression* match option is active. For the *special character* option we replace the special characters with the following regular expression ".?" and apply the above regular expression technique.

The stemming operation is language specific. GalaTex uses Galaxbuilt-in stemmer implementation, which is Porter's English stemmer [96]. The stemmer reduces the English words to their word stems. For example, the word *connections* would be reduced to its stem *connect*.

Stop words are reflected in the implementation of *FTWindow* and *FTDistance* primitives. More precisely, these primitives skip stop words when specified. The remaining match options deal with language specifics and character encoding problems. Their implementation is still underway.

3.3.3 Full-Text Scoring

In this section, we describe our implementation of a scoring technique in GalaTex. Recall that the specification of XQuery Full-Text [114] does not mandate a specific scoring method. Rather, it defines some requirements on score values based on the relevance of query answers to a full-text expression (see Section 3.2.2).

The probabilistic relational algebra is a well-established scoring method in Information Retrieval (IR) [105, 76]. This algebra operates on tuples with a score attribute. The score of a tuple represents the probability that a tuple contains a word. A score formula is associated with each algebraic operator which transforms its input tuples scores into output tuples scores. Since each *FTSelection* in our language can be viewed as an algebra operator as illustrated in the query plan in

Figure 3.2, we propose a natural adaptation of the probabilistic scoring method to *AllMatches* and show that it preserves the scoring requirements given in Section 3.2.

We first add a *score* field to the position structure described in Section 3.3.1 to capture the score of individual positions. This corresponds to adding a score to each entry in the inverted list. Conceptually, the score of an entry represents the probability that the entry contains a given word. Hence, the score value should be a float between 0 and 1. This value can be computed using techniques such as *tf* (term frequency) and *idf* (inverse document frequency) [108].

In order to compute the score of query answers, we associate a score formula with each *FTSelection*. Each formula guarantees that answers will have a score value between 0 and 1. The composition of multiple formulas in a query plan still preserves that property.

- *FTWord* builds an *AllMatches* where each match is assigned the score of the corresponding entry in the input inverted list.

- *FTAnd(AM$_1$, AM$_2$)*: Given a match m_1 in $\texttt{AM}_1$ with score s_1, a match m_2 in $\texttt{AM}_2$ with score s_2 and an output match m_3 that contains m_1 and m_2, the score s_3 of m_3 is: $s_3 = s_1 \times s_2$. This formula is similar to the one used for Boolean AND the probabilistic relational algebra and preserves the fact that the score of tuples has to be a value between 0 and 1.

- *FTOr(AM$_1$, AM$_2$)*: Given a match m_1 in $\texttt{AM}_1$ with score s_1, a match m_2 in $\texttt{AM}_2$ with score s_2 and an output match m_3 that contains m_1 and m_2, the score s_3 of m_3 is: $s_3 = 1 - (1 - s1) \times (1 - s2)$. If m_3 contains only m_1 or m_2, its score will be equal to that of m_1 or m_2.

- *FTDistance* and *FTWindow* accept an *AllMatches* AM as input and return an *AllMatches*. Given a match m in AM with score s, if m satisfies the *FTSelection* then its score s' is: $s' = s \times f$ where f is a function associated with the *FTSelection* and evaluates to a value between 0 and 1. For example, the function associated with *FTDistance(AM, dist)* is: $distance(m, dist)$ is $f = 1 - s/dist$.

Given a query plan, the final *AllMatches* carries the scores of each match. In order to score a query answer (i.e., an XML node in the evaluation context), we compose the scores of those matches that are contained in that XML node. The composition formula is similar to the one used for *FTOr*. One could use other composition formulas such as *max*.

Note that we do not have specific scoring formulas associated with *FTNegation*, *FTOrder*, *FTScope* and *FTTimes*. In our framework, these operators do not modify the scores of their input tuples. One could devise scoring formulas for each one of them (e.g., *FTTimes* could rely on the number of occurrences of a word for scoring). One interesting direction is approximate matching. For example, if two matches do not satisfy a distance, they might be returned with a lower score.

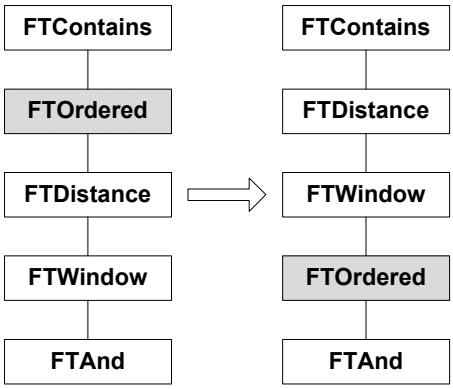
3.4 Evaluation Strategies

Our strategy to implement the XQuery Full-Text language using XML and XQuery is general and expedient, but not very efficient. In this section, we explore improvements to the current query evaluation strategies. We divide this section into improvements on full-text search and improvements on full-text scoring.

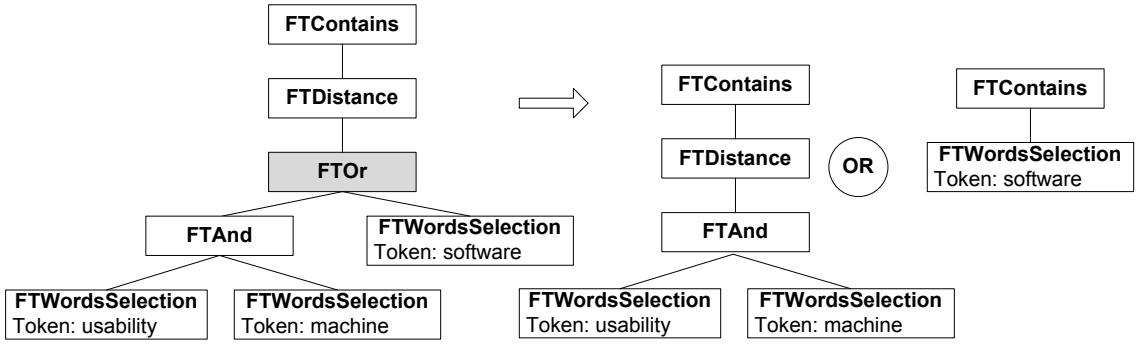
3.4.1 Full-Text Search

Given a query evaluation plan, an obvious optimization would be to push any of the primitives (*FTOrdered*, *FTDistance*, *FTWindow*, *FTScope*, *FTTimes*) as far down in the evaluation tree based on their selectivity. This is akin to pushing selections in the relational algebra. Figure 3.6(a) shows pushing *FTOrdered*. Another rewriting is short-circuiting the evaluation of *FTOr* by translating into an XQuery "or" (see Figure 3.6(b)). If one of the resulting branches evaluates to `true`, there is no need to evaluate the other one.

Nodes in an evaluation context might be structurally related, i.e., some might be descendants of others. One could organize nodes in the evaluation context in a way that guarantees that the smallest number of context nodes are checked against *AllMatches*. A node could be marked as an answer if it contains another node that



(a) FTOrder rewriting



(b) FTOr rewriting

Figure 3.6: Logical rewritings

has already been marked as an answer. This would avoid checking that node against *AllMatches*.

Materializing *AllMatches* at each step of a query evaluation tree is one of the main performance bottlenecks when evaluating queries. Pipelining, a well-studied query evaluation method in databases, would reduce the size of materialized intermediate *AllMatches*. This strategy is used in Quark,[4] which implements the TeXQuery language [24]. All our full-text primitives, except *FTTimes*, are non-blocking (i.e., they permit full pipelining of matches in *AllMatches*). *FTTimes* is partially blocking since it needs to materialize a certain number of matches. Given n search keywords (*searchToken_1, ..., searchToken_N*), the pipeline query evaluation algorithm is as follows:

[4] *http://www.cs.cornell.edu/database/Quark*

```
$EC <- XQuery/XPath
for $pos1 in getNextPosition_SortMerge(unmarked($EC), $searchToken_1)
...
for $posN in getNextPosition_SortMerge(unmarked($EC), $searchToken_N)
{
  result <- applyPrimitives($pos1,..., $posN)
  //check ancestor-descendant relationship
  //by computing the least common ancestor (LCA):
  if !empty(result) lca=LCA($pos1,..., $posN)
  if !empty(lca) markNodes($EC, lca)
  //stop condition:
  if succeeded in marking new nodes then break OR
  if allNodesMarked($EC) then break
}
output Boolean result or the marked nodes in $EC
```

We could apply the same pipelining idea to nodes in the evaluation context (i.e., to produce one node at a time). This requires pipelining the execution of the XQuery engine which might not always be possible depending on the engine that is being used. In the last case, not all matches would need to be materialized for a given context node. Figure 3.7 illustrates pipelining both nodes in the evaluation context and *AllMatches* for the query given below.

```
book[.//p ftcontains "usability" && "software"
     with distance at most 10 words]
```

A more ambitious strategy is fully integrating the XQuery Full Text data model and operators into an XQuery engine. Our strategy to implement each *FTSelection* as an XQuery function permitted rapid prototyping, but unfortunately, semantic functions do not scale, nor do they permit the flexibility required to do query optimization techniques across multiple *FTSelections*. Evolving GalaTex into a scalable implementation is not simple. Taking the step to a fully optimized evaluation strategy requires having fine-grained algebra operators that can be manipulated and composed with an XQuery algebra in order to optimize the integration

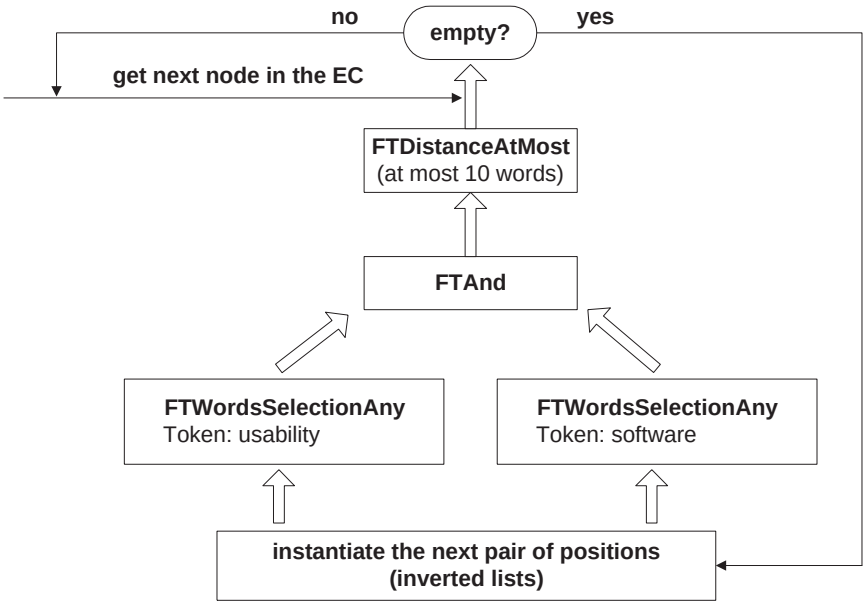


Figure 3.7: Pipelining Algorithm

on queries on structure and text. This requires integrating the *AllMatches* data model with scoring information into the XQuery data model. In particular, one can see that in order to achieve early pruning for top-k queries, we need to be able to push scoring information into an XQuery algebra. How this is achieved is an open question. However, we believe that adapting the probabilistic relational algebra for scoring as explained in Section 3.3.3 is a first step towards this integration. This would also enable scoring on both structure and content as in [28].

3.4.2 Full-Text Scoring

In Section 3.3.3, we showed that in the current GalaTex implementation, in order to score an answer (i.e., a node in the evaluation context), we produce an *AllMatches* that carries a score for each one of the matches that it contains. This means that we need to materialize all the matches in the *AllMatches* produced by a full-text evaluation plan. In the previous section, we discussed a pipelining approach to reduce the need to materialize all intermediate results. This conflicts with the need to materialize *AllMatches* to compute final answer scores. In order to implement scoring and still benefit from pipelining, we could estimate upper-bounds on the scores of those matches that have not been materialized and on the number

Table 3.1: Classification of existing IR engines for XML

IR engines	XML query engine	Search primitives	Weighting on query terms	Similarity operator	Scoring
XQuery Full-Text [114] (GalaTex)	XQuery	phrase matching, Boolean connectives, order specifier, proximity distance, no. occurrences, match options (stemming, regular expressions, stop words, case sensitive)	yes	implicit	probabilistic model or vector space model
XIRQL [75] (HyREX)	XQL	phrase matching, Boolean connectives, *$sounds_like$* operator	yes (query terms and document terms)	textual and context	probabilistic model
Flexible XML Search [118] (XXL)	XML-QL	phrase matching, limited Boolean connectives, *LIKE* operator	no	textual and context (similarity join)	probabilistic model
ELIXIR [51]	XML-QL	phrase matching, limited Boolean connectives	no	textual (similarity join)	vector space model
JuruXML [46]	Juru	phrase matching, limited Boolean connectives (negation)	no	implicit, textual and context	vector space model

of matches that a node might have in order to compute its scores without having to materialize all its matches.

3.5 Related Work and Conclusion

In database and IR research, several languages have been proposed for processing XML data on structure and text. The main focus was put on extending existing XML query languages with full-text search. However, unlike XQuery Full-Text, previous solutions explore only a few full-text search primitives at a time (e.g., Boolean keyword retrieval [73, 128], keyword similarity [51, 118], proximity distance [45], relevance ranking [44, 46, 51, 75, 83, 118]). Further, previous techniques did not provide a seamless integration with XQuery which permits querying both structure and text. More importantly, these approaches did not develop a fully composable model for full-text search the way *AllMatches* does for example.

Various ranking models have been proposed for XML in the IR literature, including the vector space model [107] and probabilistic models [105, 45]. These models provide a systematic way to compute the relevance of a document to a query. Recently, some of these models have been adapted to incorporate document structure

into account when ranking query answers. This has been the main focus of the proposals submitted to INEX [9]. In particular, XIRQL [75] and XXL [118] extend the probabilistic model while JuruXML [46] and ELIXIR [51] extend the vector space model.

Table 3.1 classifies existing XML full-text search proposals according to available search primitives and scoring techniques. Many IR engines for XML extend existing XML query engines as in the second column. We can see in this table that GalaTex fills a gap in the space of expressiveness of query languages for XML. Some of these languages incorporate explicit or implicit textual and context (element names) similarity operators used in the ranking mechanism. Most of them have decided to include limited XPath navigation in the input query and allow SQL-like queries (ELIXIR, XXL, XIRQL). Other languages have considered a more simple and intuitive query syntax by either specifying the query as an XML fragment (JuruXML) or in a Google-like style through a list of pairs: element name and keyword (XSearch). There are different approaches on the granularity of query output. XXL and ELIXIR are able to return document fragments. On the contrary, XIRQL and JuruXML focus more on relevance-oriented search and let the engine decide what nodes to return.

In this chapter, we discussed the implementation of the XQuery Full-Text language [114], an extension of XQuery language [113] that provides fully composable full-text search primitives. The TeXQuery language [24] is the main precursor of XQuery Full-Text [114]. We presented GalaTex the first conformance implementation of XQuery Full-Text that is able to query XML documents both on structure and text content. GalaTex uses XML and XQuery to implement XQuery Full text, which permits implementation on top of any existing XQuery engine. One interesting direction is to explore the use of an existing IR engine to implement some *FTSelection* and benefit from IR scoring techniques for relevance ranking.

3.6 Acknowledgements

Chapter 3, in part, is a revised reprint of the following materials published in the SIGMOD XIME-P 2005 Workshop and WWW 2005 Special interest tracks

and posters, which are joint works with Sihem Amer-Yahia, Philip Brown, and Mary Fernandez [61, 62]. The dissertation author was the primary investigator and author of the papers.

Chapter 4

Optimization of XML Full-Text Query Evaluation at Publishers

4.1 Need for Efficient Full-Text Evaluation of XML Queries using Element Nesting

There has been a flurry of recent efforts in designing languages for XML search and evaluating full-text primitives efficiently due to the increasing number of XML documents available for search. Such languages range from simple Boolean search such as [55, 80, 92] to sophisticated full-text primitives combined with XQuery and XPath [75, 118, 120, 114]. However, there is very little work on the efficient evaluation of such queries. Most query evaluation algorithms focused on the simple Boolean case and very little attention has been geared towards the evaluation of a richer class of queries with full-text predicates such as keyword proximity and order. This chapter focuses on the efficient evaluation of such predicates and their composition which involve complex proximity and order predicates on keywords. Our solution is based on exploiting element nesting to avoid redundant computation of full-text predicates on XML documents and a more efficient management of term matches by using a lazy match materialization.

More precisely, our approach relies on a unified treatment of the optimization of a large class of XML query languages with full-text predicates. The approach is based on translating them to an algebra which supports rewriting optimizations,

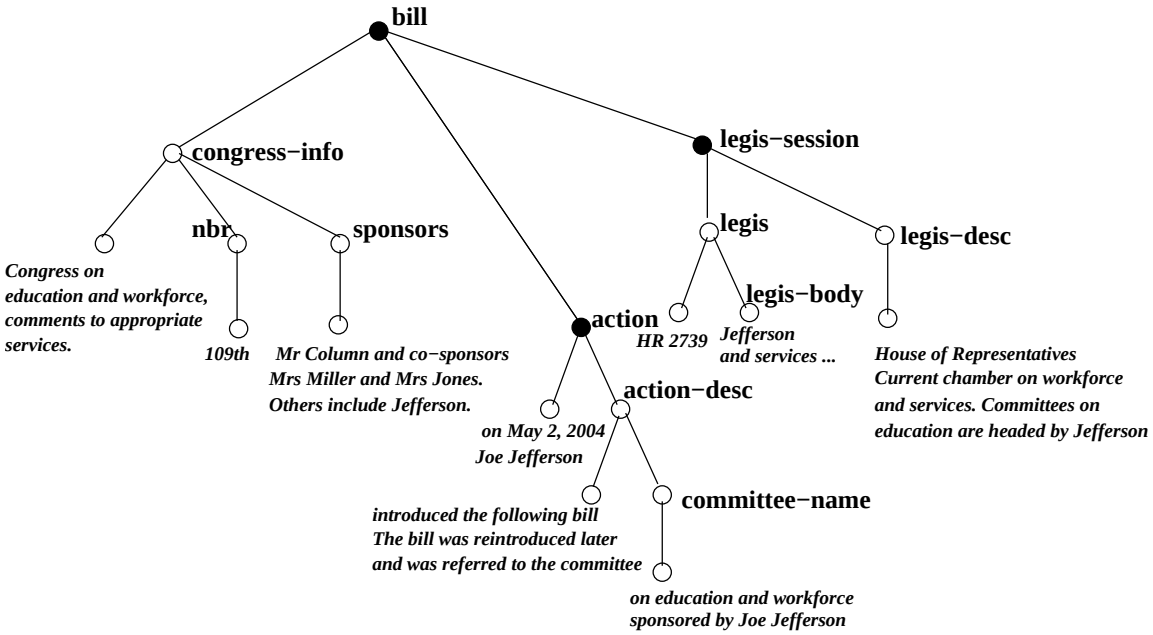
score computation and efficient evaluation, by combining relational set-at-a-time processing techniques with XML-specific exploitation of the nested document structure.

Complex full-text predicates are needed to meet the expressivity demands of increasingly sophisticated XML search applications such as digital libraries [11], and more recently, the Initiative for the Evaluation of XML retrieval methods (INEX) [9], a TREC [18]-like effort for XML. The challenges to a unified treatment of XML full-text search are posed by the variety of expressive powers and semantics. Queries range from simple keyword search to a complex combination of full-text predicates and operate on the textual content of leaf elements in the XML document tree, returning elements satisfying the predicates. Figure 4.1 shows a fragment of an XML document extracted from the Library of Congress collection [11]. A typical query would look for

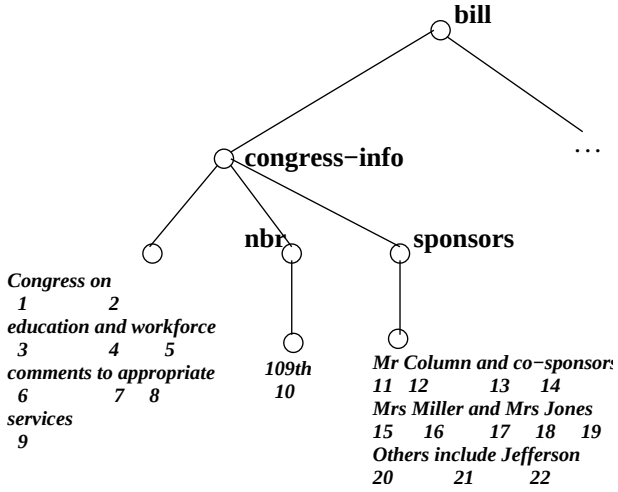
> *all elements containing the terms* `Jefferson` *and* `education` *within a window of 10 words, with* `Jefferson` *ordered before* `education` [120, 114].

Existing languages may return the most specific [80, 92] or all elements satisfying full-text predicates, possibly filtered by a user-specified structured query [75, 118, 120, 114]. For a given answer full-text predicates are checked against occurrences of query terms (also called *matches*) that belong to that answer. Predicates are usually interpreted in one of two ways. Under *binding semantics*, the same match within an answer must satisfy all query predicates. Under *existential semantics*, query predicates may be satisfied by different term matches within an answer.

Full-text predicates pose a challenge to efficient evaluation. Due to element nesting in XML documents, evaluating predicates on each element independently may result in redundant work, as matches nested within an element must be considered again when evaluating the predicates on its ancestors. The challenge is to compute the smallest necessary number of elements and matches and use element nesting to infer qualifying answers. This problem may seem simple at first since such solutions have been proposed in the past for conjunctive keyword queries (no predicates) by computing least common ancestors (LCAs) [80, 92, 109, 125]. However, due to the interplay between expressive full-text predicates which necessitate to keep track of matches, and flexible query semantics, a direct application of existing solutions would not suffice.



(a) Example XML Document



(b) Example Term Positions

Figure 4.1: XML Document for Example 4.1.1

EXAMPLE 4.1.1. *Consider the query given above on the document on Figure 4.1. Under binding semantics of the predicates, the query returns the empty answer, since whenever a match for* `Jefferson` *precedes* `education`*, they are too far apart.*

A reasonable relaxation of the query involves the more flexible existential semantics, which yields the nodes shown in solid circles in Figure 4.1(a). Each node is

selected because it contains one match of the query terms which satisfies the order predicate and another one which satisfies the distance. Prior work [80, 92, 109, 125] focuses on queries which specify only conjunctions of terms, for which the most specific answers are sought. These are the LCAs of the elements in which individual query terms occur. In our example, the set of LCAs is `congress-info`, `action`, `action-desc`, `committee-name`, `legis-session` *and* `legis-desc`. *The natural extension to predicate support corresponds to computing with each LCA only the matches which are not nested within a descendant LCA, as this compressed representation suffices to infer all matches. In our example, this means that we only keep matches contained in* `action` *which are not in* `action-desc`. *If we now apply the distance predicate to the LCA set, we obtain* `action-desc`, `committee-name`, `legis-session` *and* `legis-desc` *since none of* `congress-info` *or* `action` *contains a valid pair of matches. The order predicate will then result in* `legis-session`. *However, if we apply the order predicate first, we obtain* `action` *and* `legis-session` *and then* `action` *only after applying distance.*

Both results are incomplete: `action` *is missing in the first result and* `legis-session` *in the second. This is due to the fact that matches are only kept within their most specific element and may be filtered by the predicates.* ◇

One could envision two query evaluation strategies which compensate for the problem illustrated by Example 4.1.1. First, one could keep with each LCA all query term matches in its subtree, even those nested within descendant LCAs. This would lead to redundant computation and defeat the purpose of working with LCAs in the first place. An alternative fix would simply apply all full-text predicates simultaneously to each node. However, we would like to devise query evaluation algorithms that work regardless of the order in which predicates are applied, guaranteeing full compositionality of the predicates and with conditions requiring the match of term conjunctions, disjunctions and negations. We therefore face the challenge of integrating efficient LCA computation with efficient match manipulation (violated by the first fix), while guaranteeing such properties as commutativity, reordering and free compositionality for the full-text predicates (violated by the second fix).

The ability to evaluate full-text predicates in an order-independent manner is an important requirement to enable logical query optimization. We discuss a few

query rewritings that need to be preserved by our proposed evaluation algorithms.

So far, we have argued that devising a structure-aware algorithm improves the performance of full-text queries by reducing the amount of work needed to select qualifying XML elements. Our experiments lead us to the conclusion that match management has a high impact on the performance of full-text predicates. Informally, a match is defined as a tuple consisting of positions where the query keywords occur in an XML element. The number of such occurrences increases with the number of query terms and the size of the textual content at the document leaves. It is thus natural to think of devising evaluation strategies which minimize the time needed to materialize intermediate matches in the same spirit as relational optimizations. Furthermore, given the existential nature of full-text predicates, finding one witness match which satisfies the predicate suffices to output its containing element as an answer. We will thus see how a lazy match materialization helps improve query performance.

Finally, ranking in our context must account for different query semantics and guarantee the highest scores for the most relevant answers. Existing relevance ranking methods discussed in the related work, do not take query predicates into account. We propose to account for full-text predicates when scoring query answers and show that element scores can be computed incrementally from their descendant elements without compromising query performance. Moreover, our scoring method guarantees that answer scores are preserved among equivalent queries. This is not always true when other scoring methods are applied.

In summary, this chapter makes the following contributions:

- We introduce a formalization of XML full-text queries in terms of keyword pattern matches and we present an algebra called XFT which constructs and manipulates pattern matches using conjunction, disjunction and difference operators, as well as selection with complex full-text predicates.

  XFT permits flexible query semantics by supporting binding and existential interpretations of full-text predicates. Thus, most existing full-text languages can be expressed in XFT, which enables a uniform treatment of their evalua-

tion and optimization problems.[1] The XFT operators are freely composable, enabling query rewriting based on algebraic equivalences in the spirit of relational algebra optimization. Finally, XFT can be seamlessly integrated with algebras for structured XML search such as [23, 122], thereby enabling the optimization of queries which combine structured and full-text predicates [44, 55, 75, 118, 120, 114, 128].

- To show the feasibility of efficient evaluation of XFT expressions, we devise an algorithm called SCU (for Smallest Containing Unit), which minimizes the number of elements and matches it needs to compute at each operator in order to evaluate all query answers. Our algorithm combines relational query evaluation techniques with the stack-based exploitation of element nesting. While stack algorithms have been widely employed for LCA computation [36, 80, 92, 125], they do not straightforwardly apply to our setting; they violate our compositionality requirement by consuming input sorted according to the preorder traversal of the document and producing postorder-sorted output. The SCU algorithm takes a novel approach which transforms postorder-sorted input into postorder-sorted output. This is made possible by the off-line generation of inverted lists. Besides ensuring compositionality, this facilitates predicate evaluation since the outcome of the test on the ancestor LCAs depends on the descendant LCAs, which must hence be tested first.

- We discuss match management, another issue with significant impact on the performance of full-text predicates and devise a lazy materialization strategy inspired from relational optimizations to improve query response time.

- XFT operators enable the definition of scoring methods that account for the satisfaction of query predicates and can thus incorporate flexible query semantics (binding and existential). We show how element scores can be computed incrementally from their descendants, without compromising the complexity of evaluating the XFT operators.

[1] Translation into XFT also yields semantics specifications which are significantly more concise than the standard-provided ones (see Section 4.2.4.)

- We conduct extensive experiments to study the trade-offs implied by the use of element nesting to evaluate full-text predicates and the performance benefits of a more efficient match management. Our experiments show performance results of the SCU algorithm. Moreover, they confirm that accounting for element nesting to evaluate complex full-text predicates improves query response time by several orders of magnitude when compared to a naive evaluation of the algebra. Our performance results also compare favorably to GalaTex [61], the reference implementation of XQuery Full-Text [114].

The chapter is organized as follows. Section 4.2 presents the XFT algebra, its formal semantics, some algebraic equivalences, and the translation of XQFT [114] and NEXI [120] into XFT. It also shows how XFT can incorporate answer scoring. Section 4.3 describes the XFT evaluation algorithms. The performance evaluation of our algorithms is reported in Section 4.4. We present related work in Section 4.5 and conclude in Section 4.6.

4.2 The XFT Algebra

Formalization of full-text languages. We start from the observation that typical XML full-text languages (we shall call their family the *XQFT class* after the most expressive among them, namely XQuery Full-Text [114]) have a common characteristic: their semantics can be formalized in terms of the individual matches of keyword patterns in the input document, possibly filtering them using predicates.

Patterns and Matches. In XQFT-class languages, an expression's principal role is to specify *patterns* which are tuples of terms to be simultaneously matched against the XML document. A singleton term (k) is a pattern whose *matches* are the positions at which the term appears in the document. A term position uniquely identifies a term match and preserves order and proximity information between terms in the document. For instance, the term `education` in Figure 4.1(b) for instance appears at positions 3 and 45 and 67 further in the document. The matches of a pattern $(k_1, k_2, \ldots, k_n)$ are tuples $(m_1, m_2, \ldots, m_n)$ where each m_i is a match of term k_i. Since some full-text languages allow negation (see XQFT in Section 4.2.4), an expression may in general specify an *inclusion-exclusion pattern pair* such that each

pair p has two attributes: $p.I$ and $p.E$, each holding one pattern. Intuitively, such expressions specify nodes with matches of the inclusion pattern $p.I$ but no matches of the exclusion pattern $p.E$. Due to the presence of disjunction, queries may return *sets* of inclusion-exclusion pattern pairs. For example, the XQFT expression

$$(\texttt{Jefferson \&\& education || committee}) \texttt{ \&\& } \neg\texttt{Thomas}$$

specifies the pattern pairs

I	E
(Jefferson, education)	(Thomas)
(committee)	(Thomas)

The matching table. For any XQFT-class language, we define the semantics of a query Q to be a nested table $[\![Q]\!](N, P, M)$ (called a *matching table*), where N is an XML element node, P is a pattern, and M is a set of matches. $[\![Q]\!]$ collects the matches of all patterns specified by Q as follows: For each inclusion-exclusion pattern pair p of Q and each XML element N such that N contains no matches of $p.E$, the set M of matches of $p.I$ contained in N is non-empty, and N satisfies the predicates appearing in the query, $[\![Q]\!]$ contains a tuple t where $t.N = N$, $t.P = p.I$, and $t.M = M$.

EXAMPLE 4.2.1. *The matching table for a simple query asking for all elements containing the term* `education` *on the document in Figure 4.1(a) is*

N(ode)	*P(attern)*	*M(atches)*
bill	(education)	{(3), (45), (67)}
congress-info	(education)	{(3)}
action	(education)	{(45)}
action-desc	(education)	{(45)}
committee-name	(education)	{(45)}
legis-session	(education)	{(67)}
legis-desc	(education)	{(67)}

◇

XFT algebra. We designed the XFT algebra to construct matching tables for queries in all XQFT-class languages, thus enabling a uniform treatment of their evaluation and optimization problems. XFT facilitates rewriting optimizations and lends itself to efficient evaluation using algorithms that combine relational query evaluation techniques with the exploitation of document structure to process XML queries with complex full-text predicates. Section 4.2.1 defines the XFT data model and operators which are inspired from relational algebra. In Section 4.2.2, we show relational-style algebraic rewritings which can benefit optimizations. Section 4.2.4 shows translations into XFT for the XQFT [114] and the NEXI [120] languages. Section 4.2.3 explains how XFT permits scoring of query answers under flexible query semantics.

4.2.1 XFT Operators

The XFT operators manipulate matching tables, composing them into new tables or filtering their tuples according to predicates. This is in the same spirit as the XQFT data model described in [114]. The formal semantics of XFT is shown in Figure 4.2 and detailed next. The selection operators in XFT are defined for the binding and the existential predicate semantics. R_i denote matching tables.

- **get**(k) returns a table containing one tuple t for each node n with a non-empty set of matches (denoted $matches(n, k)$) of term k against the subtree rooted at n. In Figure 4.2, $\mathcal{N}$ denotes the set containing a unique identifier for each node in the input document collection. The result of evaluating **get**(`services`) on the document on Figure 4.1 is a table containing one entry for each of `congress-info`, `bill`, `legis-body`, `legis`, `legis-session` and `legis-desc`.

- R_1 **or** R_2 returns a table which collects for each node n and pattern p, the union of the corresponding matches found in R_1 and in R_2.

- The conjunction operator R_1**and**R_2 creates a new table for the nodes with matches given by both R_1 and R_2. For each such node n, if R_i states that the matches of pattern p_i under n are the set m_i, we may infer that n actually contains matches of pattern $p_1 \circ p_2$. The operator $\circ$ concatenates two patterns,

eliminating duplicate terms. For instance, $(k_1, k_2) \circ (k_3, k_2, k_4) = (k_1, k_2, k_3, k_4)$. The empty pattern () is the identity element: $p \circ () = () \circ p = p$. All elements contain a match of (). It is easy to see that the matches of $p_1 \circ p_2$ are given by the natural join of $m_1 \bowtie m_2$.

Since the expressions delivering the operands of **and** may contain **or**, the join of the corresponding matching tables can in principle construct several distinct tuples agreeing on the N and P attributes. In this case, **group** coalesces all matches by union-ing them together.

- R_1**minus**R_2 returns the tuples in R_1 pertaining to nodes without matches in R_2.

Summarizing, operators **get**, **and**, **or** and **minus** find for each inclusion-exclusion pattern pair p specified by the expression, the nodes n with no matches of $p.E$ and the actual matches of $p.I$ under n.

The remaining operators, also referred to as full-text predicates, test various conditions on the matches (possibly filtering them in the process). For each full-text predicate P, we provide two operators P^b and $P^\exists$, to support binding, respectively existential semantics. In Figure 4.2, K denotes a pattern $k_1, \ldots, k_l$, and $\Pi_K(M)$ the projection of the set of matches M on the components corresponding to the terms in K. If the matches in M do not correspond to a pattern which includes the terms in K, $\Pi_K(M) = \emptyset$.

- **times**$_{\theta n}(k_1, \ldots, k_l)$ applied to an input table R, selects the tuples whose match count for patterns containing the terms $k_1, \ldots, k_l$, satisfies the θ-comparison to the integer n.

- **ordered**$^\exists(k_1, \ldots, k_l)$ applied to table R returns the tuples containing some match in which the position of term k_i appears in the document order before that of the term k_{i+1} for all i in the input term list $k_1, \ldots, k_l$. The other flavor of this operator, **ordered**$^b(k_1, \ldots, k_l)$, drops from each tuple t the matches in $t.M$ which violate the above ordering condition. If all matches are dropped, so is t.

- $\textbf{window}^{\exists}_{\theta n}(k_1, \ldots, k_l)$ applied to an input table R, returns the tuples $t \in R$ containing some match which fulfills the window condition, i.e. in which all positions of terms $k_1, \ldots, k_l$, lie within a maximum distance which satisfies the θ-comparison with integer n.
  $\textbf{window}^{b}_{\theta n}(k_1, \ldots, k_l)$ drops all non-conforming matches first (dropping also t if no matches are left).

- $\textbf{dist}^{\exists}_{\theta n}(k_1, \ldots, k_l)$ applied to an input table R, returns the tuples $t \in R$ containing some match which conforms to the condition that the positions of terms which are adjacent in the pattern $(k_1, \ldots, k_l)$, are at distance satisfying the θ-comparison with integer n.
  $\textbf{dist}^{b}_{\theta n}(k1, \ldots, k_l)$ drops all non-conforming matches first (and also t if no matches are left).

EXAMPLE 4.2.2. *The query in Example 4.1.1 is expressible in* XFT *as (we abbreviate the terms for readability):*

$$\sigma_{\textbf{\textit{ordered}}^{\exists}(\texttt{Jeff},\texttt{edu})}(\sigma_{\textbf{\textit{window}}^{\exists}_{\leq 10}(\texttt{Jeff},\texttt{edu})}(\textbf{\textit{get}}(\texttt{Jeff})\ \textbf{\textit{and get}}(\texttt{edu})))$$

Here we used $\textbf{\textit{ordered}}^{\exists}$ *and* $\textbf{\textit{window}}^{\exists}$ *to denote the existential semantics where an answer must contain at least one match of* `Jefferson` *and* `education` *that satisfies window and one possibly different match that satisfies order.* [2] ◇

We now formalize the connection between the results of XFT algebra expressions and the matches of patterns against the input XML trees. Like all XQFT-class queries, each XFT expression E specifies a set of inclusion-exclusion pattern pairs denoted with $ppairs(E)$ and defined formally in Figure 4.3. The XFT operators omitted from the figure do not affect the set of pattern pairs. The inclusion-exclusion pattern pair of the expression in Example 4.2.2 is $(I = (\texttt{Jefferson}, \texttt{education}), E = ())$ – the query contains no negation and therefore specifies no exclusion pattern.

[2] To relax the query further and return answers containing only one term, we could replace **and** by an outer join.

Operator	Definition
$\mathbf{get}(k)$	$\{t \mid n \in \mathcal{N}, M = matches(n,k), M \neq \emptyset, t.N = n, t.P = (k), t.M = M\}$
$R_1 \mathbf{or} R_2$	$\mathbf{group}(R_1 \cup R_2)$
$R_1 \mathbf{and} R_2$	$\mathbf{group}(\{t \mid t_1 \in R_1,\ t_2 \in R_2,\ t.N = t_1.N = t_2.N, t.P = t_1.P \circ t_2.P, t.M = t_1.M \bowtie t_2.M\})$
R_1 **minus** R_2	$\{t \mid t \in R_1, t.N \notin \Pi_N(R_2)\}$
$\sigma_{\mathbf{times}_{\theta n}(K)}(R)$	$\{t \mid t \in R, (\sum\{\lvert\Pi_K(t_1.M)\rvert \mid t_1 \in R, t_1.N = t.N\})\ \theta\ n\}$
$\sigma_{\mathbf{ordered}^{\exists}(K)}(R)$	$\{t \mid t \in R,\ \exists(m_1, m_2, \ldots, m_l) \in \Pi_K(t.M), m_1 < m_2 < \ldots m_l\}$
$\sigma_{\mathbf{ordered}^{b}(K)}(R)$	$\{t \mid t_1 \in R, t.N = t_1.N, t.P = t_1.P, M = \{m \mid m = (m_1, m_2, \ldots, m_l) \in \Pi_K(t_1.M), m_1 < m_2 < \ldots m_l\}, M \neq \emptyset, t.M = M\}$
$\sigma_{\mathbf{window}^{\exists}_{\theta n}(K)}(R)$	$\{t \mid t \in R,\ \exists(m_1, m_2, \ldots, m_l) \in \Pi_K(t.M), (\max_{1 \leq i,j \leq l} distance(m_i, m_j))\ \theta n\ \}$
$\sigma_{\mathbf{window}^{b}_{\theta n}(K)}(R)$	$\{t \mid t_1 \in R, t.N = t_1.N, t.P = t_1.P, M = \{m \mid m = (m_1, m_2, \ldots, m_l) \in \Pi_K(t.M), (\max_{1 \leq i,j \leq l} distance(m_i, m_j))\theta n\}, M \neq \emptyset, t.M = M\}$
$\sigma_{\mathbf{dist}^{\exists}_{\theta n}(K)}(R)$	$\{t \mid t \in R,\ (m_1, m_2, \ldots, m_l) \in \Pi_K(t.M), \bigwedge_{1 \leq i < l} distance(m_i, m_{i+1})\ \theta\ n\}$
$\sigma_{\mathbf{dist}^{b}_{\theta n}(K)}(R)$	$\{t \mid t_1 \in R, t.N = t_1.N, t.P = t_1.P, M = \{m \mid m = (m_1, m_2, \ldots, m_l) \in \Pi_K(t.M), \bigwedge_{1 \leq i < l} distance(m_i, m_{i+1})\ \theta\ n\}, M \neq \emptyset, t.M = M\}$
$\mathbf{group}(R)$	$\{t \mid t_1 \in R, t.N = t_1.N, t.P = t_1.P, t.M = \bigcup\{t_2.M \mid t_2 \in R, t_2.N = t_1.N, t_2.P = t_1.P\}, t.M \neq \emptyset\}$
θ	$\in \{=, <, \leq, >, \geq\}$

Figure 4.2: The XFT Algebra

THEOREM 4.2.1. *Let E be an* XFT *algebra expression consisting only of* ***get***, ***and***, ***or*** *and* ***minus*** *operators. Then the result of E on any collection of XML documents is a matching table R such that $t \in R$ if and only if there is some pattern pair $pp \in ppairs(E)$ where*

(i) $t.P = pp.I$,

(ii) $t.M$ holds all matches of $pp.I$ under $t.N$, and

(iii) there are no matches of $pp.E$ under $t.N$. ◇

$$
\begin{array}{rcl}
ppairs(\textbf{get}(k)) & = & \{p \mid p.I = (k), p.E = ()\} \\
ppairs(E1 \textbf{and} E2) & = & \{p \mid p_1 \in ppairs(E1), \\
 & & \quad p_2 \in ppairs(E2), \\
 & & \quad p.I = p_1.I \circ p_2.I, \\
 & & \quad p.E = p_1.E \circ p_2.E\} \\
ppairs(E1 \textbf{or} E2) & = & ppairs(E1) \cup ppairs(E2) \\
ppairs(\neg E) & = & \bigcup\{\{p_1, p_2\} \mid p \in ppairs(E), \\
 & & \quad p_1.E = p.I, p_1.I = (), \\
 & & \quad p_2.E = (), p_2.I = p.E()\}
\end{array}
$$

$$(p \circ () = () \circ p = p)$$

Figure 4.3: Inclusion-Exclusion pairs of patterns in XFT expressions

4.2.2 XFT Rewriting Optimization

The main benefit of using an algebra is to be able to apply logical rewritings to queries while preserving the set of query answers. The design of XFT is highly influenced by the relational algebra in order to enable typical rewritings such as pushing selections and join reordering. While a full-fledged rewriting-based optimizer is subject of future work, our experiments already confirm the expected performance benefit of rewriting-based optimization (Section 4.4).

The formal semantics of XFT implies a plethora of algebraic equivalences. We only list a few here that we implemented as a proof of concept. Let us consider again the query in Example 4.2.2. We have seen that its expression in the algebra is (we abbreviate the terms for readability):

$$\sigma_{\textbf{ordered}^{\exists}(\texttt{Jeff},\texttt{edu})}(\sigma_{\textbf{window}^{\exists}_{\leq 10}(\texttt{Jeff},\texttt{edu})}(\textbf{get}(\texttt{Jeff}) \textbf{ and } \textbf{get}(\texttt{edu})))$$

This expression is equivalent to

$$\sigma_{\textbf{ordered}^{\exists}(\texttt{Jeff},\texttt{edu}) \,\wedge\, \textbf{window}^{\exists}_{\leq 10}(\texttt{Jeff},\texttt{edu})}(\textbf{get}(\texttt{Jeff}) \textbf{ and } \textbf{get}(\texttt{edu}))$$

and to

$$\sigma_{\textbf{window}^{\exists}_{\leq 10}(\texttt{Jeff},\texttt{edu})}(\sigma_{\textbf{ordered}^{\exists}(\texttt{Jeff},\texttt{edu})}(\textbf{get}(\texttt{Jeff})\ \textbf{and}\ \textbf{get}(\texttt{edu})))$$

Now consider adding to this query the condition that the term `services` appear after `education` in the text. This query could be written as

$$\sigma_{\textbf{ordered}^{\exists}}(\texttt{Jeff},\texttt{edu})(\sigma_{\textbf{ordered}^{\exists}(\texttt{edu},\texttt{services})}(\sigma_{\textbf{window}^{\exists}_{\leq 10}(\texttt{Jeff},\texttt{edu})}(\textbf{get}(\texttt{Jeff})\ \textbf{and}\ \textbf{get}(\texttt{edu})\ \textbf{and}\ \textbf{get}(\texttt{services})))),$$

which is equivalent after pushing selections into the **and** operator, to

$$\sigma_{\textbf{ordered}^{\exists}}(\texttt{Jeff},\texttt{edu})(\sigma_{\textbf{window}^{\exists}_{\leq 10}(\texttt{Jeff},\texttt{edu})}(get(\texttt{Jeff})\ \textbf{and}\ \textbf{get}(\texttt{edu})) \textbf{and}\ (\sigma_{\textbf{ordered}^{\exists}(\texttt{edu},\texttt{services})}(\textbf{get}(\texttt{edu})\ \textbf{and}\ \textbf{get}(\texttt{services}))))$$

Finding LCAs of query terms is a typical implementation of the **and** operator which has been widely used in previous work [80, 92, 109, 125]. However, as we have shown in Example 4.1.1, simply computing LCAs and applying selection predicates does not always return the set of correct answers due to violating the above equivalences.

Section 4.3 shows our solution to combining the best of two worlds in algorithms that preserve query rewritings and enable the implementation of the **and** operator using LCAs.

4.2.3 Scoring

The goal of scoring in the context of XFT is twofold: (i) allow to manipulate scored answers in a query plan while preserving the query semantics and, (ii) define answer scores in a way that guarantees that the score of an element node can be computed efficiently. We describe a scoring method that conforms to these requirements.

We assume that scores are stored along with tuples in the initial matching tables and define term weights as follows:

Term Weights. We adapt the standard *tf*idf* function [79, 55] to individual

nodes and compute the weight of a term k for a given leaf node n. This function is defined as: (i) *idf*, or inverse document frequency, that quantifies the relative importance of an individual term in the collection of documents; and (ii) *tf*, or term frequency, that quantifies the relative importance of a term in an individual document. In the vector space model [32], query terms are assumed to be independent of each other, and the *tf*idf* contribution of each term is added to compute the final score of the answer document. Intuitively, since XML queries return elements as opposed to whole documents, the weight of a term in elements of different types (tag) may be different. We denote *itf*, the *idf* of elements of the same type.

The term frequency *tf(n,k)* is defined as ($occ(k, n)$ denotes the number of distinct occurrences of term k in leaf node n):

$$tf(n,k) = \frac{occ(k,n)}{max\{occ(k',n) \mid k' \in words(n)\}}$$

Let T be the set of all nodes that share the same tag as node n, then, *itf(n,k)* is defined as:

$$itf(n,k) = \log(1 + \frac{\mid T \mid}{\mid \{n \in T \mid k \in words(n)\} \mid})$$

Intuitively, `bill` elements have a different relevance to a given term than `committee-info` elements. The fewer there are elements of the same type, the higher their *itf* for a term.

We note that the *tf* of a node for a term can be inferred from its descendant nodes while *itf* needs to be pre-computed and stored with the node. The **get** operator in our algebra could be used to retrieve term weights. Next, we define an element score.

Answer Scores. We define the weight of a term k in an answer s as

$$\texttt{tf(s,k)} * \texttt{itf(s,k)}$$

Given a pattern containing a set of keywords $\mathcal{K}$, the score of an answer is defined as:

$$\texttt{score}(\texttt{s}) = \sum_{\texttt{k} \in \texttt{K}} (\texttt{tf}(\texttt{s}, \texttt{k}) \times \texttt{itf}(\texttt{s}, \texttt{k}))$$

Given this definition, it is easy to see that **and** can compute the score of each output tuple using the above formula while **or** and **minus** would only need to preserve input scores. Binding predicates may choose to modify scores in a way that is different from existential ones thereby enforcing query semantics when computing scores. Consequently, the use of an algebra permits to better control intermediate answer scores and decide whether or not individual query operators have an impact on scores.

Similarly to vector-based scoring, our method assumes independence between term weights within an element node. Thus, a key advantage of our scoring method is the ability to compute the score of a node in an incremental fashion from its descendant nodes without affecting the algorithms complexity. More sophisticated scoring is possible (though with higher evaluation complexity) if the independence assumption is relaxed, as in probabilistic IR models [76]. In this chapter, we focus on independent scoring functions.

4.2.4 Application to XQFT and NEXI

XQFT. XQuery Full-Text (XQFT), an upcoming W3C standard [114], is an extension of XPath and XQuery to allow with full-text predicates. Wherever XQuery allows a predicate, XQFT allows the expression **ftcontains**(E) with E a text search expression. For example,

$$\begin{aligned} &/books/book[review \textbf{ ftcontains} \\ &\quad ((\text{“}thumbs\text{”} \&\& \text{“}up\text{”} \,||\, \text{“}must\text{”} \&\& \text{“}read\text{”}) \textbf{ distance} \leq 1 \\ &\quad ||(\text{“}best\text{-}seller\text{”} \textbf{ times} \geq 2))]/author \end{aligned}$$

returns all authors of books whose very enthusiastic review contains either the term pair (“thumbs”,“up”) in close proximity or similarly the pair (“must”,”read”) , or mentions the term “best-seller” at least twice.

$$
\begin{array}{rcl}
E & \rightarrow & \text{“}k\text{”} \\
 & | & E_1 \&\& E_2 \\
 & | & E_1 \,||\, E_2 \\
 & | & E \ \textbf{times}\theta n \\
 & | & E \ \textbf{ordered} \\
 & | & E \ \textbf{window}\theta n \\
 & | & E \ \textbf{distance}\theta n \\
 & | & (E) \\
 & | & \neg E \\
\theta & \rightarrow & = \ | \ < \ | \ \leq \ | \ > \ | \ \geq \\
k & \rightarrow & \textit{any term}
\end{array}
$$

Figure 4.4: Syntax of W3C's Standard XQFT

$$
\begin{array}{rcl}
[\![k]\!] & = & \textbf{get}(k) \\
[\![E_1 \&\& E_2]\!] & = & [\![E_1]\!] \ \textbf{and} \ [\![E_2]\!] \\
[\![E_1 || E_2]\!] & = & [\![E_1]\!] \ \textbf{or} [\![E_2]\!] \\
[\![E \ \textbf{times} \ \theta n]\!] & = & \textbf{times}_{\theta n}([\![E]\!]) \\
[\![E \ \textbf{ordered}]\!] & = & \textbf{ordered}^b([\![E]\!]) \\
[\![E \ \textbf{window}\theta \ n]\!] & = & \textbf{window}^b_{\theta n}([\![E]\!]) \\
[\![E \ \textbf{distance} \ \theta n]\!] & = & \textbf{dist}^b_{\theta n}([\![E]\!]) \\
[\![(E)]\!] & = & [\![E]\!] \\
[\![\neg E]\!] & = & \{t \mid n \in \mathcal{N}, t.N = n, t.P = (), t.M = \{()\}\} \\
 & & \textbf{minus} \ [\![E]\!]
\end{array}
$$

Figure 4.5: Specification of XQFT Semantics in XFT

The syntax of the XQFT expressions that may appear in the scope of the **ftcontains** term is given in Figure 4.4. The XQFT primitives not shown in the figure do not affect the patterns as they do not mention any terms. We define the binding semantics of XQFT in Figure 4.5.

NEXI. Similarly to XQFT, we show how our algebra captures the semantics of NEXI [120], the language that is being used within INEX [9] to express XML search queries. The core expression in NEXI is the **about** expression which permits

conjunction of query terms. For example,

/books/book[**about**(*review*, *"thumbs" "up" "must" "read" "best-seller"*)]/*author*

returns book authors whose review *is about* one of the terms. NEXI also allows the specification of weights which is not yet supported in XFT. The syntax of NEXI is very simple: $E \rightarrow$ "$k''|E_1\ E_2$. We express its semantics through translation into XFT: $[\![\text{"}k'']\!] = \textbf{get}(k)$, $[\![E_1E_2]\!] = [\![E_1]\!]$ **or** $[\![E_2]\!]$.

4.3 XFT Evaluation Algorithms

Algorithms to compute LCAs of query terms have shown very good performance for the evaluation of conjunctive keyword queries [80, 92, 109, 125]. We have seen in Section 4.1 that their applicability to the evaluation of full-text predicates is not straightforward. This section presents novel algorithms that implement operators in the algebra proposed in Section 4.2. Section 4.3.1 describes AllNodes a straightforward implementation of our algebra. More efficient algorithms that use element nesting, are presented in Section 4.3.2. We finish with a brief note on incremental scoring in SCU.

4.3.1 The AllNodes Algorithm

A key design goal of the XFT algebra was amenability to efficient, set-at-a-time pattern match construction and filtering by leveraging tried-and-true relational techniques. Indeed, notice in Figure 4.2 that R_1**and**R_2 joins R_1 and R_2 on N, aggregating the M attributes of joining tuples via natural join; R_1**or**R_2 corresponds to taking the union of R_1 and R_2, grouping it by the N and P attributes, and aggregating the M attributes by union-ing them; both flavors of **ordered**, **window** and **distance** are simple selections over the M attribute of each tuple, and **times** can be evaluated by grouping on N followed by aggregation using the count function, with a `having` clause for the θ-comparison.

We have implemented this evaluation strategy as a proof of concept in the AllNodes algorithm detailed next.

Node and Word Identifiers. We choose to represent node and term match identifiers using the well-established Dewey encoding which enables efficient computation of the LCA of two nodes (or term matches) as the longest common prefix of their ids [80, 92, 109, 125]. For example, the id of the first match of the term `services` encountered in the document in Figure 4.1(b) is 1.1.1.9 where 1 is the id of the root node `bill`, 1.1 is the id of node `congress-info` and 1.1.1 is the id of the `text` node containing the term. Also, testing whether node a is an ancestor of node d reduces to finding a's id as a prefix of d's id.

The get operator. The **get** operator is implemented using a lookup in a standard inverted list index [32] IL which associates with each term k the list of its matches, given by their Dewey ids. The list of matches is stored in top-down, right-child-first traversal order. **get**(k) needs to return all nodes containing the match of k, but the inverted lists only store *the immediately containing node*, i.e., leaf nodes in the XML document. While this requires some processing when reading the inverted lists, the alternative of storing all ancestor nodes of a term match is known to lead to tremendous space overhead and is commonly avoided [80]. The processing required to restore all ancestors of a term match is minimal: we obtain their ids as the strict prefixes of the id of the match. The implementation of **get** must

1. perform a single pass over the input inverted lists.
2. collect for each node all term matches under it before outputting the node;
3. avoid duplicate output of node ids.

We satisfy all these desiderata using Algorithm 4.1 below.

Algorithm 4.1 get(k) in AllNodes Implementation

Require: inverted list IL stores matches of k in top-down, rightmost-child-first traversal order

Ensure: outputs matching table sorted in descending lexicographic order on N

```
1: initialize stack
2: push(t), where t.N = the root, t.P = (k), t.M = ∅
3: (m, n) = get_next_match(IL, k)        ▷ m is the match of k, n its immediately containing node
4: while (stack not empty) do
5:     t=top(stack)
6:     if n is a descendant-or-self of t.N then
7:         add m to all tuples in stack
8:         for (each proper descendant d of t.N which is an ancestor-or-self of n) do
9:             push(t_d), where t_d.N = d, t_d.P = (k), t_d.M = {m}        ▷ push higher nodes first!
10:        end for
11:        (n,m) = get_next_match(IL, k)
12:    else
13:        output t and pop the stack
14:    end if
15: end while
```

The algorithm produces the output sorted by the descending lexicographic order of the Dewey ids of the N attribute. Thus, the task of subsequent operators is to preserve this order to enable merge-style algorithms, as detailed below.

The and operator. The **and** implementation performs a merge of the two (already sorted) inputs. Whenever tuples t_1 and t_2 have the same value for their N attribute, the set of matches of the resulting tuple t is computed by taking the natural join of $t_1.M$ with $t_2.M$. This is in keeping with Theorem 4.2.1, since all matches of the combined pattern under $t.N$ are indeed found this way. The results of the joins can be large, degenerating in most cases (when the patterns in t_1 and t_2 do not overlap) to Cartesian products. These are notoriously expensive to compute, both in terms of time and space. This computation can be performed lazily, sometimes avoided entirely, and in most cases preceded by a reduction of the operand sizes. The idea is to record the two operands of the natural join (worst-case Cartesian product) in $t.M$ without further computation. To this end, $t_1.M, t_2.M, t.M$ hold *lists* of sets of matches, with the understanding that a list represents the multi-way join of all its sets, in the order they are listed. We describe below how these matches

are materialized when evaluating full-text predicates.

The or operator. **or** essentially unions its operands R_1 and R_2, grouping the result by the N and P attributes and union-ing together the matches in each group. The union is computed by simply merging the operands, which are already sorted on N, breaking ties by sorting on P. The merge ensures that the output is ordered on N, P as well. The patterns are compared using lexicographic order induced by the alphabetic order of the individual terms. The grouping involves no additional overhead, as the operands are already sorted on the grouping attributes so each group is listed contiguously in the merge of R_1 and R_2.

The times operator. **times** needs to group its operand table on N, counting the matches in each group. This requires a linear scan, since the table is already sorted on N.

The Other Operators. All other operators are order-preserving, as they at most drop tuples from the table. They require one linear scan of the input table.

Avoiding Cartesian product. Recall that the **and** operator does not materialize matches, instead simply computing the list of $t.M$ as the append of the list $t_2.M$ and the list $t_1.M$. As long as the result of this **and** operator is consumed by other **and** operators, matches won't need to be materialized and the lists keep growing. Materialization is delayed until required by a predicate, at which point the Cartesian product is pruned by pushing selections into it, using equivalences in the spirit of those illustrated in Section 4.2.2.

Pipelining. All our operator implementations can be pipelined (the **group** operator used to implement **and** and **or** is not fully blocking, as its input is sorted).

4.3.2 The SCU Algorithm

The definition of matching tables was primarily introduced with the purpose of providing a clean formal semantics of XML full-text search which captures various query language semantics, in particular XQFT (as shown in Section 4.2.4). Matching tables also enable relational-style evaluation of full-text queries, as demonstrated in Section 4.3.1.

However, matching tables are not necessarily the ideal data structure to ma-

nipulate during evaluation. Indeed, by definition of the matching table, whenever a table R contains a tuple t, R also contains a tuple t_a for each ancestor a of $t.N$, and $t.M$ is included in $t_a.M$. $t.M$ is therefore redundantly stored in R, forcing the AllNodes algorithm (especially the full-text predicates) to repeatedly visit its contents. The redundancy increases with element nesting, i.e., the depth of node t_d in the document.

In this section, we introduce an alternative evaluation algorithm that operates on tables which eliminate precisely the redundant storage of the descendant's matches in the ancestor. These tables, called SCU tables (for Smallest Containing Unit), lead to smaller intermediate results and therefore to potentially better overall performance. Our experiments (Section 4.4) show that this potential is indeed reached.

DEFINITION 4.3.1. (**SCU tables**) SCU tables have the same schema as matching tables, but satisfy:

1. for every pair of tuples t_a, t_d such that $t_a.N$ is an ancestor of $t_d.N$ and $t_a.P = t_d.P$, the descendant's matches are not repeated in the ancestor's: $t_a.M \cap t_d.M = \emptyset$. We call $t_a.M$ the *direct* matches of node $t_a.N$ and $t_d.M$ *inherited* matches of node $t_a.N$.

2. there is no tuple t_a such that $t_a.M = \emptyset$ and such that

$$|\{t_d \mid t_d.P = t_a.P,\ t_d \text{ is descendant of } t_a\}| = 1$$

◇

An immediate implication on SCU tables is that in any tuple t, $t.N$ is the LCA of all matches in $t.M$.

EXAMPLE 4.3.1. *Recall the matching table of the term* `education` *in Example 4.2.1. Its corresponding* SCU *table is given below, this time revealing the Dewey ids of the nodes and positions:*

N	P	M
1.1	(education)	{(1.1.1.3)}
1.2.2.2	(education)	{(1.2.2.2.1.45)}
1.3.2	(education)	{(1.3.2.1.67)}

◇

SCU tables contain the same information as matching tables, but in a compressed way. Any matching table can be reduced into an SCU table: for each tuple t_a, remove from $t_a.M$ the set $t_d.M$ whenever $t_d.N$ is a descendant of $t_a.N$ and $t_a.P = t_d.P$, and drop t_a altogether if $t_a.M$ becomes empty and if there is precisely one tuple t_d with $t_d.P = t_a.P$ and where $t_d.N$ is a descendant of $t_a.N$. We call this operation `reduce`. Its inverse is called `expand` and it turns an SCU table into a matching table in the obvious way.

The SCU algorithm takes an XFT expression E and a list of SCU tables $S_1, \ldots, S_n$ and returns an SCU table $SCU_E(S_1, \ldots, S_n)$. The SCU algorithm is semantically equivalent to the XFT algebra, in the sense that for any matching tables $R_1, \ldots, R_n$ and any XFT expression E,

$$\begin{array}{c} E(R_1, \ldots, R_n) \\ = \qquad\qquad\qquad\qquad (\dagger) \\ \texttt{expand}(SCU_E(\texttt{reduce}(R_1), \ldots, \texttt{reduce}(R_n))). \end{array}$$

SCU tables are possible due to the XML tree structure since they rely on element nesting information stored in Dewey ids to expand into matching tables. SCU tables lead to a significant simplification in the implementation of the **get** operator, which now needs to output for each term match only the immediately containing node but none of the ancestors. The implementation of the **ordered**^{b}, **dist**^{b}, **window**^{b} operators is not affected, as all they do is filter matches, dropping the ones who violate the filtering condition regardless of whether they are stored redundantly. There is no change of semantics if these full-text predicates work on SCU tables instead of matching tables. However, the runtime performance is improved as fewer tuples are inspected, and shorter lists of matches per tuple are scanned.

The conciseness of SCU tables comes at a price though: it poses new problems

to the evaluation of the other full-text predicates, as well as the **and** operator, in the sense that directly applying their AllNodes implementation to SCU tables breaks requirement (†), as detailed below.

and can no longer be computed as an equijoin on the N attribute, as illustrated by the following example.

EXAMPLE 4.3.2. *Consider the* XFT *expression*

get(`Column`) ***and get***(`introduced`)

For the document in Figure 4.1(a), there is only one match of the term `Column` *and of* `introduced`*. The immediately containing nodes are* `sponsors`*, respectively* `action-desc`*. Let* $R_{\texttt{Column}}$ *and* $R_{\texttt{introduced}}$ *be the* SCU *tables returned by* ***get***(`Column`) *and* ***get***(`introduced`) *according to the* SCU *implementation. These contain one tuple each,* $t_{\texttt{Column}} \in R_{\texttt{Column}}$ *with* $t_{\texttt{Column}}.N =$ `sponsors`*, and similarly for* $t_{\texttt{introduced}} \in R_{\texttt{introduced}}$*. The LCA* `bill` *of* `sponsors` *and* `action-desc` *contains a match of pattern* (`Column`, `introduced`)*, but the equi-join of* $t_{\texttt{Column}}$ *with* $t_{\texttt{introduced}}$ *is empty, failing to produce the corresponding tuple. The* `expand` *operation cannot remedy the problem, as the expansion of an empty table remains empty. In contrast, by Theorem 4.2.1, since* (`Column`, `introduced`) *has a match under* `bill`*,* ***and*** *should output such a tuple as is indeed the case in the* AllNodes *implementation.* ◇

The **ordered**$^{\exists}$,**dist**$^{\exists}$, **window**$^{\exists}$ and **times** operators are affected as well if operating on SCU tables. The semantics of full-text predicates depends on *all* matches appearing under a node, which is why in the matching table, each tuple t is *self-contained* for the purposes of predicate evaluation: all matches under node $t.N$ are collected in $t.M$. The full-text predicates can therefore be evaluated locally on each tuple, *akin to predicates in relational selections*. In contrast, an SCU table keeps the matches relevant to the evaluation of a full-text predicate on node $t_a.N$ *distributed across several tuples* corresponding to descendants of $t_a.N$. In this case, full-text search predicates are turned into global aggregation operations working on the entire table. This is illustrated in Example 4.1.1. We summarize it here.

EXAMPLE 4.3.3. *Recall that the query in Example 4.1.1 is*

$$\sigma_{\textbf{\textit{ordered}}^{\exists}(\texttt{Jeff},\texttt{edu})}(\sigma_{\textbf{\textit{window}}^{\exists}_{\leq 10}(\texttt{Jeff},\texttt{edu})}(\textbf{\textit{get}}(\texttt{Jeff})\textbf{\textit{ and get}}(\texttt{edu})))$$

Node `action` *is returned since it contains one match for each of the query terms satisfying* ***ordered***$^{\exists}$ *and one satisfying* ***window***$^{\exists}$*. However, since the match satisfying* ***ordered***$^{\exists}$ *(resp.,* ***window***$^{\exists}$*) violates* ***window***$^{\exists}$ *(resp.,* ***ordered***$^{\exists}$*), each match would be filtered prematurely regardless of the order of predicate application.* ◇

We compensate for these problems by adapting the implementation of the affected operators as follows.

The and operator. The previous example shows that the equi-join does not work on SCU tables, and that two tuples $t_1 \in R_1, t_2 \in R_2$, despite not agreeing on their N attribute, carry matches relevant to the LCA of $t_1.N$ and $t_2.N$. This suggests an immediate fix: when "joining" tuples t_1 and t_2, output the tuple t with $t.N = LCA(t_1.N, t_2.N), t.P = t_1.P \circ t_2.P, t.M = t_1.M \bowtie t_2.M$.

This approach poses significant efficiency challenges. Since any two nodes from the same document have an LCA (the document root in the worst case), the tempting but naive implementation involves a Cartesian product. A more efficient alternative would be to adopt one of the state-of-the-art stack-based algorithms developed in prior work to evaluate conjunctive keyword searches in XML documents by computing the LCAs of the matches, without keeping track of the matches themselves [125], or without evaluating any full-text predicates on them [36, 80].

Preorder versus Postorder Incompatibility. The immediate adaptation of existing stack-based algorithms to our needs is precluded by the fact that existing algorithms assume the input sorted in *preorder*, but produce their result in *postorder*. This is not a problem in prior works, which focus only on computing LCAs of a conjunctive patterns without post-processing the result. In our setting this mismatch precludes operator compositionality.

Compositional Stack-Based Algorithms. We propose a stack-based solution yielding an efficient single-pass algorithm which computes S_1 **and** S_2, where S_1, S_2 are SCU tables sorted in postorder traversal order. The result is also sorted in postorder, thus facilitating the seamless composition of **and** operators with each

other (and, as we shall see below, with all other operators) without any intervening sorting step. This is important for evaluation performance, and essential for enabling pipelined implementation. Though stack-based processing is not a new idea, the solution for consuming input in postorder is novel and guarantees efficient compositionality of our algebra operators.

Algorithm 4.2 (on page 138) uses a stack containing descendant LCAs and their matches for consideration by ancestor LCAs.

The stack holds tuples of schema

$$(L, Dir_1, Dir_2)$$

and the algorithm maintains the invariant that, according to the input consumed so far,

- L is an LCA of at least one pair of matches from $S_1 \times S_2$;
- Dir_i is a set of matches (called the *direct matches*) from S_i, each contained in L but occurring in no LCA which is a proper descendant of L;
- The LCAs in the L attribute of the stack entries reside on the same root-to-leaf path, with the deepest LCA at the top of the stack. This is achieved by pushing a newly computed LCA only if it is a descendant of the stack top's LCA, $top().L$.

The stack is maintained while two cursors are advanced in a single pass over the input SCU tables S_1 and S_2, reading tuples s_1, s_2 respectively.

If the new LCA l computed at line 20 is greater in postorder than $top().L$ (line 22 in Algorithm 4.2), the postorder sorting of the inputs guarantees that no further descendant LCAs of $top().L$ can be encountered and we can pop and output the latter (lines 26–27). Additionally, if the newly computed LCA l is not an ancestor of $top().L$, there is a new LCA l' induced by $top().L$ and l (computed at line 24), and l' must be pushed (line 41) on the stack before l (line 44), to maintain the descendant-last invariant for stack LCAs. Notice that l' has no direct matches, since they are all nested within l and $top().L$.

If the new input contributes to the same LCA as $top().L$ (line 30), this input is recorded in the top stack tuple (lines 30–36), since we need to accumulate all direct matches contributing to the LCAs.

Finally, the new input can generate an LCA l which is a descendant of $top().L$. At this point (line 36), we know that the input matches are not direct matches for $top().L$, and they must be removed from the $top().Dir_i$ lists (lines 37–40). Moreover, l is pushed on the stack since now we expect the remaining input to contribute to its descendant LCAs (line 44).

When a stack tuple o is output, we generate SCU table tuples t from it. This involves setting $t.N = o.L$ and computing the matches corresponding to the pairs in the Cartesian product $o.Dir_1 \times o.Dir_2$ (not shown in the pseudocode).

The node operations performed by Algorithm 4.2 are all very well supported by Dewey ids. Indeed, checking that l is larger than $top().L$ in postorder (line 22) reduces to checking that the Dewey id of l is either lexicographically larger than or a strict prefix of the Dewey id of $top().L$. Similarly for the tests in line 14. The ancestor test in line 26 reduces to testing that the Dewey id of l' is a strict prefix of $top().L$'s id. The LCA computations are implemented as simply finding the longest common prefix of the operands.

Algorithm 4.2 SCU Implementation of **and** Operator - Part 1

Require: S_1, S_2 are SCU tables sorted in postorder on N

Ensure: outputs SCU table corresponding to `reduce`(`expand`(S_1) **and** `expand`(S_2)) sorted in postorder on N attribute

1: initialize stack
2: open cursors on S_1 and on S_2
3: $s_1 \leftarrow$ get_next(S_1), $s_2 \leftarrow$ get_next(S_2)
4: $l \leftarrow LCA(s_1.N, s_2.N)$
5: push($L = l, Dir_1 = \{s_1\}, Dir_2 = \{s_2\}$)
6: **while** (at least one cursor can advance) **do**
7: **if** ($EOF(S_2)$) **then**
8: $s_1 \leftarrow$ get_next(S_1);
9: **else if** ($EOF(S_1)$) **then**
10: $s_2 \leftarrow$ get_next(S_2)
11: **else**
12: $s_1' \leftarrow$ look_ahead(S_1); $l_1 \leftarrow LCA(s_1'.N, s_2.N)$
13: $s_2' \leftarrow$ look_ahead(S_2); $l_2 \leftarrow LCA(s_1.N, s_2'.N)$
14: **if** (($l_1 <_{post} l_2$ or $l_1 = l_2$ and $s_1.N \leq_{post} s_2.N$) **then**
15: $s_1 \leftarrow$ get_next(S_1)
16: **else**
17: $s_2 \leftarrow$ get_next(S_2)
18: **end if**
19: **end if**
20: $l \leftarrow LCA(s_1.N, s_2.N)$
21: $l' \leftarrow l$
22: **if** ($l >_{post} top().L$) **then**
23: **if** (l is not ancestor of $top().L$) **then**
24: $l' \leftarrow LCA(l, top().L)$
25: **end if**
26: **while** (l' is ancestor of $top().L$) **do**
27: $o \leftarrow$ pop(); output o

Algorithm 4.3 SCU Implementation of **and** Operator - Part 2

```
28:         end while
29:     end if
30:     if (l' = top().L) then
31:         if (S_1 cursor was last to advance) then
32:             top().Dir_1 ← top().Dir_1 ∪ {s_1}
33:         else
34:             top().Dir_2 ← top().Dir_2 ∪ {s_2}
35:         end if
36:     else
37:         if (non-empty stack) then
38:             top().Dir_1 ← top().Dir_1 \ {s_1}
39:             top().Dir_2 ← top().Dir_2 \ {s_2}
40:         end if
41:         if (l' ≠ l) then
42:             push(L = l', Dir_1 = ∅, Dir_2 = ∅)
43:         end if
44:         push(L = l, Dir_1 = {s_1}, Dir_2 = {s_2})
45:     end if
46: end while
47: while (non-empty stack) do
48:     o ← pop(); output o
49: end while
```

Evaluation of Full-text Predicates. Since each SCU tuple contains only the direct matches under its nodes, but the predicates depend also on indirect matches, their evaluation needs to fulfill the following requirements:

- detect that a descendant already satisfied the predicate and hence the ancestor's matches needn't be tested
- if a descendant tuple t_d does not satisfy the predicate, before dropping t_d its direct matches must be propagated to the tuple t_a of its immediate ancestor. The matches are needed, as they might satisfy predicate operators higher up in the plan.

These requirements suggest a natural evaluation strategy, which exploits and preserves the postorder sorting of the inputs, leading to full compositionality of all operators. The strategy calls for considering descendants first, using a stack to propagate matches to ancestors, as detailed in Algorithm 4.4 (on page 141). The stack

tuples have schema (T, D) where T is an SCU tuple and D a boolean flag. The algorithm maintains the invariant that D is set to true iff the predicate is satisfied by $T.N$ or any of its descendants consumed so far from the input. When a new tuple s is read, if $s.N$ is an ancestor of the node at the stack top $top().T.N$, we cannot expect further input to contribute any descendants of $top().T.N$, and it is safe to pop (line 16). If the popped D flag is set to true, we record that s satisfies the predicate (line 11) through its indirect matches, so the predicate need not be checked on the direct matches of s (line 18). Otherwise, we drop the descendant (but not before propagating its matches to s (line 14)) and we check the predicate on s (line 19). If s satisfies the predicate either through direct or indirect matches, it is output. Either way, s is pushed on the stack (line 24) together with the verdict on the predicate's satisfaction, for subsequent consumption by ancestors of $s.N$.

Algorithm 4.4 SCU Implementation of Full-Text Predicates

Require: S is an SCU table sorted in postorder on N the predicate $P \in \{\textbf{ordered}^{\exists}, \textbf{window}^{\exists}, \textbf{dist}^{\exists}\}$ applies to individual SCU tuples

Ensure: outputs SCU table corresponding to $\texttt{reduce}(\sigma_P(\texttt{expand}(S)))$ sorted in postorder on N

```
 1: initialize stack; open cursor on S
 2: s ← get_next(S); satisfies ← P(s)
 3: if (satisfies = true) then
 4:     output s
 5: end if
 6: push(s, satisfies)
 7: while (not EOF(S)) do
 8:     s ← get_next(S); satisfies ← false
 9:     while (s.N is ancestor of top().T.N) do
10:         if (top().D = true) then
11:             satisfies = true
12:         else
13:             s.M ← s.M ∪ top().T.M          ▷ propagate matches to ancestor
14:         end if
15:         pop()
16:     end while
17:     if (satisfies = false) then      ▷ only check predicate if descendants violate
18:         satisfies ← P(s)
19:     end if
20:     if (satisfies = true) then
21:         output s
22:     end if
23:     push(T = s, D = satisfies)
24: end while
```

EXAMPLE 4.3.4. *The query plan on Figure 4.6 illustrates the input and output* SCU *tables of Example 4.1.1. For instance, node* `legis-session` *(Dewey* 1.3*) is selected due to the propagation of its* `Jefferson` *match* 1.3.2.1.72 *by the* $\textbf{ordered}^{\exists}$ *predicate. The final set of answers is* `action`, `legis-session` *and* `bill`. ◇

Algorithms Complexity. For each tuple consumed from the input, Algorithm 4.2 performs constant-time stack manipulation operations, computes an LCA (which depends on the length of the common prefix of the Dewey ids), and later, upon popping the tuple, it generates matches. When applied to inputs $|S_1|$ and $|S_2$ and producing output $|S|$, the running time contains: a linear component in the size of the inputs $|S_1| + |S_2|$; a linear component in the size of the output $|S|$; denoting with D

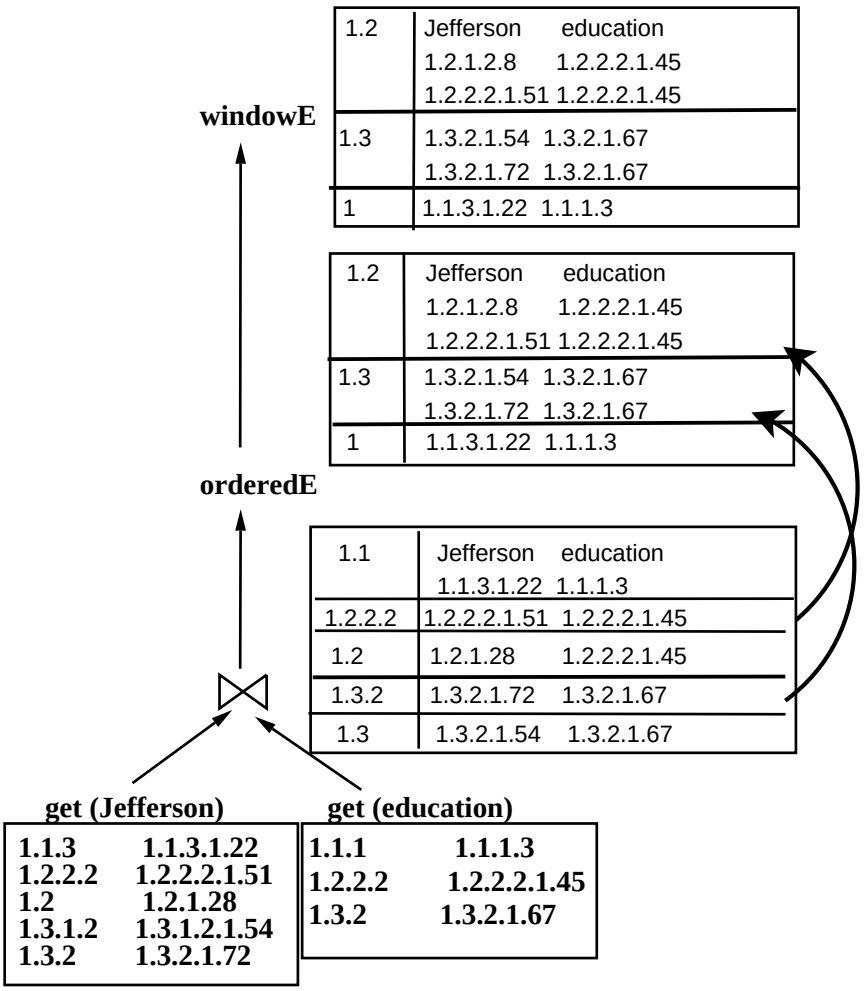


Figure 4.6: SCU Execution for Query in Example 4.1.1

the maximum nesting depth of a discovered LCA, it performs for each LCA at most D operations of scanning the Dewey id. The total number of LCAs is upper-bounded by the smallest size among $|S_1|$ and $|S_2|$: $O(|S_1| + |S_2| + |S| + D \times min(|S_1|, |S_2|)$. Algorithm 4.4 runs in time worst-case linear in the size of the input, $|S|$, if the predicate is not satisfied and all matches end up being inspected and propagated up the XML hierarchy.

Scoring. It is easy to see that the use of a stack in Algorithm 4.4 provides direct access to the descendants of a node, found at the top of the stack when the node gets pushed. This is compatible with the incremental computation of the score of a node from its descendant nodes. The node's score can be easily updated/recomputed

as long as the scoring function depends on descendants only.

4.3.3 Match Management in SCU Operators

We now discuss the management of matches in the evaluation of the SCU **and** operator and the full-text predicates. For the sake of presentation simplicity, these subtle tasks were not detailed in the pseudocode of Algorithms 4.2 and 4.4.

Eager match materialization. A direct implementation of the match management in the operators according to the specification of their semantics in Section 4.2.1 would manipulate matches as follows. SCU tables would indeed represent for each tuple t in the table the set of matches $t.M$ associated to node $t.N$ as a set of tuples, one tuple per match. Upon consuming two SCU tuples t_a and t_b from its inputs, the **and** operator would, after some stack manipulations, finally output a new tuple o whose node $o.N$ is the LCA of $t_a.N$ and $t_b.N$, and whose set of matches contain the Cartesian product of $t_a.M$ and $t_b.M$ (making sure that only direct matches are stored). The straightforwardness of this implementation strategy comes at a price, namely an explosion in the number of matches in $o.M$. In general, the subplan corresponding to a conjunctive query of k keywords, each occurring an average of c times in the document, would in the worst case yield nodes with $O(c^k)$ matches. We call this strategy the *eager* match materialization strategy, since it materializes matches within the **and** operators, before these matches are even needed by the predicates.

Lazy match materialization. In our implementation, we adopted the *lazy* match materialization alternative, based on two key observations. First is a classic insight from relational query evaluation: avoiding the large intermediate results yielded by Cartesian products by turning sequences of products followed by selections into joins. Second, we observe that predicates need not test all matches of a given node: finding one witness match which satisfies the predicate suffices to output the node.

Consequently, we picked a representation of SCU tables which minimizes the match manipulation overhead in **and** operators, storing just enough information to enable materialization later, during predicate evaluation. At that time, we interleave

the process of match materialization with that of predicate testing, aggressively pruning match materialization steps. More specifically, in any tuple t of the SCU table we represent the matches of a pattern $(k_1, \ldots, k_n)$ only implicitly, by storing n lists: to each keyword k_i, we associate the list L_i of its matches. The matches of the entire pattern correspond to the Cartesian product of these lists, which however we do not compute in the **and** operator. Instead, match manipulation in the **and** operator reduces to simple list concatenations.

At predicate evaluation time, we conceptually need to compute the Cartesian product of the lists, then filter the obtained matches through the predicate, dropping the SCU table entry if no matches qualify. In practice, we iterate over the lists using n nested loops, each nested loop attempting to extend the partial match given by the current iteration of its outer loops. If we detect that a partial match already violates the predicate such that no extension to a full match can satisfy it, the inner loop searching for match extensions is skipped. This procedure corresponds to an n-way nested-loop join of the position lists, pushing the selection predicate into the join. This effectively prunes the materialization work for all extensions of the violating partial match. Testing that a partial match violates a predicate p involves exploiting p's specific semantics and is done on a predicate-by-predicate basis. For example, if $p = window < 5$, it suffices to find two positions in the partial match at distance higher than 5 to know that no extension of the partial match can satisfy p. Notice that the same cannot be done for $p = window \geq 5$, but the latter is less frequently used than the former version, as typical queries tend to care about keywords occurring in close proximity. Similarly, if $p = ordered$, it suffices to find two adjacent positions which are out of order in the partial match to prune any of its extensions to full matches. Similar reasoning allows us to extract pruning criteria for the remaining predicates.

The representation of matches using individual keyword position lists also speeds up match propagation. Recall from Section 4.3.2 that when all of a node's matches violate a predicate, the node's entry is dropped from the SCU table and its matches are propagated to the lowest ancestor's entry. In the chosen representation, propagation becomes trivial: migrating the matches of a pattern p to an ancestor who also has matches of pattern p consists in simply concatenating the lists corresponding

to the same keywords.

Note that for queries involving no predicates, the match materialization never happens. For queries with predicates, no materialization occurs in the predicate-free subplans. This is why in the experiments (Section 4.4), we isolate the LCA computation and stack management tasks from the match materialization task by running experiments with both predicate-free and selection queries.

4.3.4 Correctness of SCU Algorithm

We sketch the key ingredients behind the proof of correctness of the SCU algorithm, which states that the evaluation algorithm returns the result specified by the formal XFTalgebra semantics as given in Section 4.2. This statement is formalized in equation (†) in terms of the two operators `reduce` and `expand`, defined in Section 4.3.2.

We call a matching table T *reduced* if $T = \texttt{reduce}(T)$.

Given an XFT operator o, we denote with o^{SCU} its implementation according to the SCU algorithm. It is easy to see that statement (†) follows straightforwardly by induction on the structure of the XFT expression from the following statements:

Let o be an arbitrary XFT operator except for **get**. o may be unary or binary, its arity is clear from the context in the statements below. Let T, T_1, T_2 be arbitrary reduced matching tables sorted on attribute N in postorder (notice that this makes them SCU tables by definition). Then

$$o(\texttt{expand}(T)) = \texttt{expand}(o^{SCU}(T)) \tag{4.1}$$

$$o(\texttt{expand}(T_1, T_2)) = \texttt{expand}(o^{SCU}(T_1, T_2)) \tag{4.2}$$

and

$$o^{SCU}(T) \quad \text{is a reduced table sorted on } N \text{ in postorder} \tag{4.3}$$

$$o^{SCU}(T_1, T_2) \quad \text{is a reduced table sorted on } N \text{ in postorder} \tag{4.4}$$

and (for the base case of the induction)

$$\textbf{get}(k) = \texttt{expand}(\textbf{get}^{SCU}(k)) \tag{4.5}$$

We briefly sketch the steps and intuition behind the proofs of these statements.

(4.5) follows immediately from the way inverted index lists are stored, namely listing only the nodes directly containing the keywords.

(4.4) is most interesting for the case $o = \textbf{and}$. In this case, lines 20–49 in Algorithm 4.2 are responsible for maintaining the invariant detailed on page 136, which essentially states that all LCAs on the stack reside on the same root–leaf path in the tree, with the deepest LCA at the top of the stack. This results in a postorder output of the nodes. It also facilitates the direct match management for the LCA at the top of the stack whenever a new LCA is found which is its descendant. This in turn allows the removal of any LCAs without direct matches during the output operation, thus ensuring that the resulting table is reduced. Details on how the invariant is maintained by the various steps of the algorithm are given on page 136.

(4.3) pertains to Algorithm 4.4 for predicate evaluation. Because of the same invariant that the stack contains at every step a list of nodes residing on the same root-leaf path, with the deepest node on top, the order in which nodes are output is postorder. Moreover, nodes with no direct matches satisfying the predicate are popped and dropped, while their matches are propagated by simply migrating them to the new stack top which corresponds to the lowest ancestor of the popped node. This ensures that the output table is reduced.

(4.1) also refers to predicate operators, and requires us to prove the claim that Algorithm 4.4

(i) misses no node with direct matches which satisfy the predicate, and

(ii) for these nodes misses none of their direct matches, whether they satisfy the predicate or not.

Once this claim is proven, the expansion of the resulting table must yield (by definition of the `expand` operator) *all* nodes having some match which satisfies the predicate.

The claim follows from the fact that predicate evaluation does not compute any new nodes, it only drops them from the table. Dropping happens only if all direct matches violate the predicate (yielding (i)), but even then the matches are propagated to the lowest ancestor due to the stack invariant (yielding (ii)).

(4.2) for the case $o =$ **and** has the most interesting proof. Here we need to show that the SCU Algorithm 4.2 for **and** misses no LCAs with direct matches obtainable from the two input tables S_1 and S_2. If all pairs of in the Cartesian product $S_1 \times S_2$ were inspected, the claim would follow immediately, but at the cost of quadratic running time. However, the two tables are scanned linearly by advancing cursors s_1 and s_2. The cursor management is given in lines 6–19 in Algorithm 4.2. Whenever cursor s_1 is advanced to the lookahead s_1', the pairs involving s_1 and the remaining tuples in S_2 are pruned from consideration. We need to show that any LCA yielded by such a pruned pair would also be yielded by the pairs involving s_1'. We say that s_1' subsumes s_1 in that case. Showing that s_1' subsumes s_1 whenever the cursor advances from latter to former (and symmetrically for s_2' and s_2) involves a detailed case analysis on where s_1, s_1', s_2, s_2' may lie in the tree with respect to each other. Many cases are eliminated due to the fact that both S_1 and S_2 are ordered in postorder.

4.4 Experiments

We first perform a series of experiments that demonstrate the superiority of the SCU over the AllNodes algorithm (Section 4.4.2).

In Section 4.4.3, we carry out a more in-depth evaluation of the SCU algorithm, isolating the overheads due to various tasks such as the stack management, the propagation of matches to ancestor nodes, and the materialization of matches during predicate evaluation. We also demonstrate the benefit of using relational-style query rewritings such as pushing predicates into the joins.

Finally, we compare the SCU algorithm with GalaTex [61] a conformance implementation of XQuery Full-Text [114], and the TeXQuery Quark implementation [3] (Section 4.4.4).

4.4.1 Experimental Setup

Each experiment involves generating the appropriate data sets, building the corresponding indices and evaluating full-text queries over the data using various query evaluation algorithms.

The data sets.

Naturally, we investigated the scalability of our algorithms with increasing document size. Moreover, since the focus of the SCU algorithm is on exploiting the nested document structure to avoid redundant computation, we also studied its behavior when increasing document depth. To control the size and depth while using realistic data, we started from a large real dataset, namely the DBLP XML document [3], and varied the size by incrementally appending snapshots of the DBLP data. We used the November 2005 snapshot of the DBLP bibliography which lists over 700,000 articles totaling 300MB.

To isolate the effect of document size we generated a family of synthetic documents of increasing size while keeping the depth constant. Each document was generated from a chain of 10 elements nested within each other (9 single-child internal elements and one leaf element) by embedding n sibling copies of the real DBLP snapshot into the chain's leaf element, where n is an integer parameter. A nesting depth of 10 is representative of a large class of documents. In order to further understand the behavior of our algorithm, we also investigated the effect of significantly higher document depth, as described below.

We isolated the effect of document depth by generating a family of documents of increasing depth yet with constant text content. To this end, we started from a chain of depth m (with m a parameter ranging up to 70) elements with no text content, inserting one copy of the DBLP snapshot into the chain's leaf element.

[3] *http://www.cs.cornell.edu/database/quark*

We also ran experiments on the Wikipedia XML Corpus [65] by considering different snapshots of the articles in the first 550MB. The English Wikipedia has a mean document depth of about 6.72. While DBLP is relatively flat and has regular structure, Wikipedia consists of irregularly structured elements describing a web-based encyclopedia.

The queries.

To study the behavior of both algorithms with increasing query complexity, we varied the number of keywords and full-text predicates per query. The queries used in the experiments are listed below.

Table 4.1: XML full-text queries used in experiments

Q	Query description
Q_1	**get**(Alin) **and get**(Fernandez) **and get**(Alon) **and get**(Levy) **and get**(Mary)
Q_2	$\sigma_{\textbf{window}^{\exists}_{<5}(\text{Alin,Fernandez,Alon,Levy,Mary})} Q_1$
Q_3	$\sigma_{\textbf{window}^{\exists}_{>1}(\text{Alin,Fernandez,Alon,Levy,Mary})} Q_1$
Q_4	$\sigma_{\textbf{dist}^{\exists}_{>5}(\text{Alin,Fernandez,Alon,Levy,Mary})} Q_1$
Q_5	**get**(Alin) **and get**(Fernandez) **and get**(Alon) **and get**(Levy) **and get**(Mary) **and get**(Deutsch) **and get**(Daniela) **and get**(Florescu) **and get**(Dan) **and get**(Suciu) **and get**(Language)
Q_6	$\sigma_{\textbf{window}^{\exists}_{<5}(\text{Alin,Fernandez,Alon,Levy,Mary})}$(**get**(Mary) **and** $\sigma_{\textbf{window}^{\exists}_{<5}(\text{Alin,Fernandez,Alon,Levy})}$(**get**(Levy) **and** $\sigma_{\textbf{window}^{\exists}_{<5}(\text{Alin,Fernandez,Alon})}$(**get**(Alon) **and** $\sigma_{\textbf{window}^{\exists}_{<5}(\text{Alin,Fernandez})}$(**get**(Fernandez) **and get**(Alin)))))

The query term frequencies for one DBLP snapshot are shown in the table below. With multiplicity of the number of snapshots the frequencies also multiply. For example for a 10-copies snapshot of DBLP we have 10 times the frequencies specified in the table for each keyword.

The measurements.

Conforming to common practice [85], we report only the query execution times, which do not include the times to access the inverted lists since they depend

Table 4.2: Query term frequencies for DBLP snapshot

Term	DBLP snapshot (300MB)
`Alin`	64
`Alon`	651
`Dan`	2404
`Daniela`	595
`Deutsch`	144
`Fernandez`	246
`Florescu`	63
`Language`	9008
`Levy`	502
`Mary`	1757
`Suciu`	124

solely on the underlying database/storage engine backend (Berkeley DB in our case). All times are reported in milliseconds. Each value reported in our graphs is an average collected from 20 runs of the experiments.

The platform.

The algorithms were implemented in Java and the parsing of XML document collection was performed using the SAX API of the Xerces2 Java Parser [4]. The inverted lists were backed up by Berkeley DB 4.4.16 [5]. The experiments were conducted on a Centrino 2GHz laptop with 1GB of RAM running Windows XP Professional.

4.4.2 Comparison of AllNodes and SCU

Scalability with Document Nesting Depth

We study the effects of document nesting on both algorithms. The experiment confirms the following expectation: Since the SCU algorithm uses a stack-based approach to exploit the document structure during query evaluation, we expect it to scale well with increasing document depth. In contrast, the AllNodes algorithm replicates at each node the computation of matches already found in its descendant nodes. This redundant computation is expected to increase with the document

[4] http://xerces.apache.org/xerces2-j/
[5] http://www.sleepycat.com

nesting depth.

To isolate the effect of nesting depth, we used the family of XML documents of increasing depth and constant text content described in Section 4.4.1. For example, in Figure 4.7 the data point 70 corresponds to an XML document consisting of a chain of 70 single-child elements nested within each other, where the element at nesting depth 70 is the entire DBLP document. Its ancestors contain no immediate text.

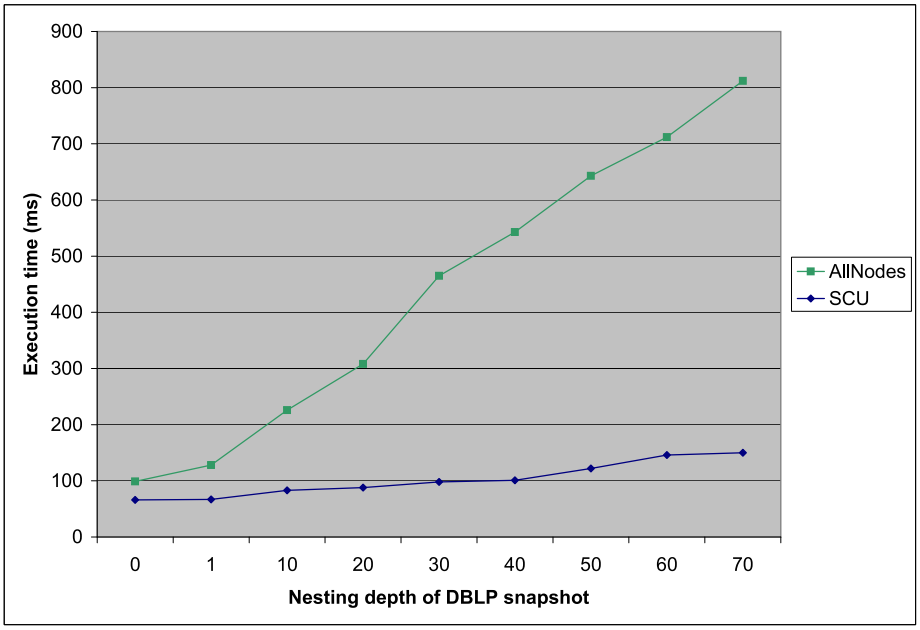


Figure 4.7: Running query Q_1 on variable-depth chain ending in DBLP snapshot

Figure 4.7 shows the result of running AllNodes and SCU on query Q_1.

Notice that even for shallow documents the SCU algorithm outperforms the AllNodes algorithm and the gap between the two increases with the document nesting depth. The dominant reason is that AllNodes handles larger intermediate results than SCU since matches contained in a node are computed again at each one of its ancestors.

We observe that the SCU algorithm is almost insensitive to the variation of the nesting depth. Even though the set of elements in the query answer grows with the document depth, SCU handles the exact same intermediate result tables on all inputs. This is because the tables are reduced and do not list any of the chain (the ancestor nodes of the DBLP snapshot), as these have no direct matches.

The small variation in SCU performance is a side effect of our particular choice of representing the node identifiers using Dewey notation: deeper documents yield longer Dewey node identifiers, which require longer manipulation time (for such operations as testing id equality or finding LCAs).

Scalability with Document Size

This experiment shows that the SCU algorithm wins even when we increase the document size without increasing the nesting depth.

Figure 4.8 depicts the results of evaluating query Q_1 according to the AllNodes and SCU algorithms on a family of documents of increasing size (ranging from 300MB to 3GB) yet of constant depth, generated as described in Section 4.4.1. For example, data point 7 in Figure 4.8 corresponds to an XML document of size 2.1GB consisting of a chain of nested nodes of depth 10, in which the node at depth 10 contains 7 children, each a copy of the 300MB DBLP snapshot.

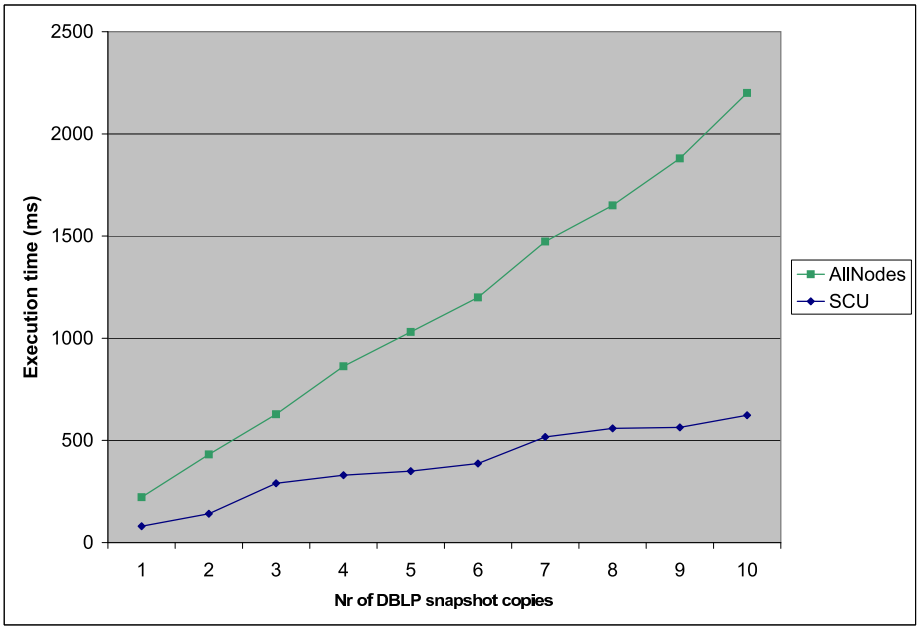


Figure 4.8: Running Q_1 on chain of depth 10 with multiple DBLP snapshots as leaves

Notice that the running times for both algorithms grow linearly with the document size. This may appear counter-intuitive, given that for a conjunctive query of k keywords, the number of matches over n copies of DBLP should increase

by a factor of n^k when compared to the number of matches over only one copy. For Q_1, $k = 5$, which is far from linear.

Recall however from Section 4.3.3 that both algorithms represent matches implicitly, by keeping for each keyword the list of its matches and delaying the computation of the Cartesian product of these lists until needed for the evaluation of a predicate. At that point, the Cartesian product can be pruned using the predicate, thus effectively turning into a join. In this experiment, the query Q_1 contains no predicates. Therefore, increasing the number of DBLP copies results in a merely linear increase in the length of each keyword's position lists. The overhead of actually materializing matches in the presence of predicates is measured in Section 4.4.3.

Scalability with Query Complexity

Since a conjunctive query with k terms translates into a k-way join, the following experiments evaluate the performance of both algorithms for increasing k.

To this end, we use a family of queries obtained from the prefixes of query Q_5, with the shortest query containing the first two terms and the longest query Q_5 itself, containing 11 terms, a large number chosen to stress the implementation. Because of the crucial impact of the document nesting depth, we measure the running time of these queries in two extreme cases: for the original 300MB DBLP snapshot, and for the version nested within a chain of depth 70.

Shallow documents. Figure 4.9 reports the running times for this family of queries on the original 300MB DBLP document, which we call the "shallow" DBLP version.

The absence of nesting in the flat DBLP document favors the AllNodes algorithm, whose running time is close to that of the SCU algorithm. Nevertheless, SCU outperforms AllNodes even in this case.

The sudden jump in the execution time when the query is extended from 10 to 11 terms is due to the low selectivity of term nr. 11 ("Language"), which leads to an explosion of matches of the extended pattern.

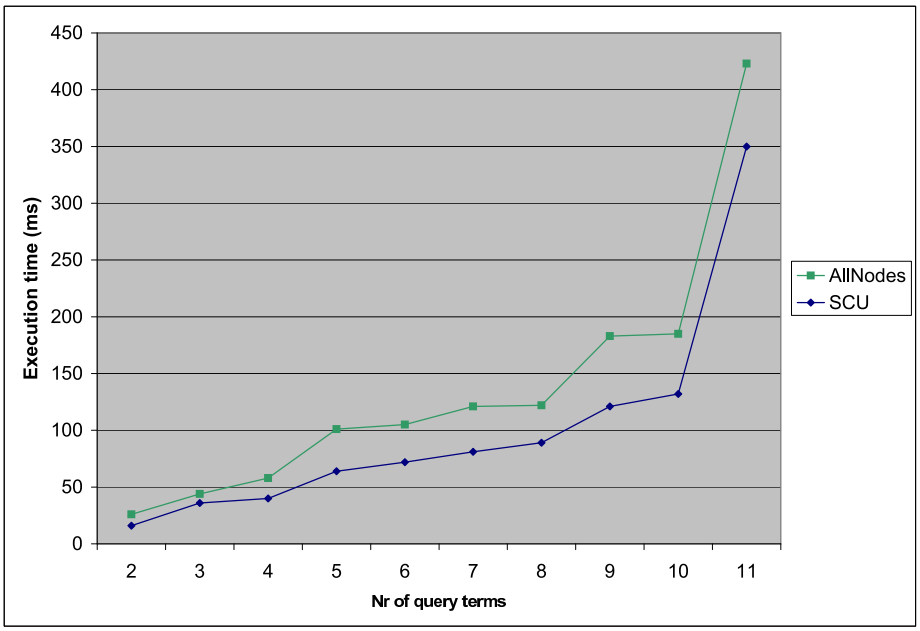


Figure 4.9: Running prefixes of query Q_5 on the shallow DBLP document

Deep documents. Figure 4.10 reports the running times for the same prefix queries of Q_5 on an extremely nested XML document, namely the 70-level depth chain 300MB DBLP.

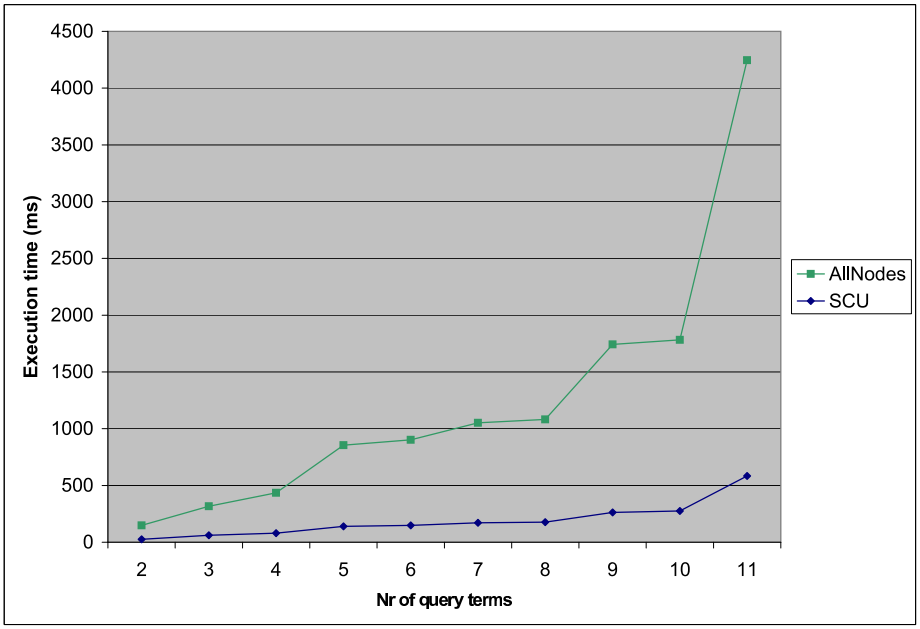


Figure 4.10: Running prefixes of Q_5 on DBLP snapshot nested in chain of depth 70

As predicted, SCU outperforms AllNodes and the performance gap between the two increases more dramatically because SCU is less sensitive to document depth.

Real data set: Wikipedia XML.

To validate our findings against real data, we ran query Q_7 under both implementations against Wikipedia [65] snapshots of increasing size (we concatenated the many small documents in the Wikipedia collection into two documents, of sizes 250MB and 500MB).

Q	Query description
Q_7	**get**(`carrier`) **and get**(`small`) **and get**(`same`) **and get**(`iron`) **and get**(`water`)

The frequencies of the terms used in Q_7 are listed below.

Term	250MB	550MB
`carrier`	495	1016
`small`	5771	12283
`same`	8809	18225
`iron`	781	1483
`water`	3845	8934

This experiment proves that SCU algorithm performs better than AllNodes on real data sets as well.

Figures 4.11 and 4.12 confirm the same trends we have observed in the previous experiments, namely a linear increase of running time with the document size for both SCU and AllNodes.

The time difference between runs of different Wikipedia snapshots is explained by the increase in the query term frequencies between the 250MB and the 550MB document.

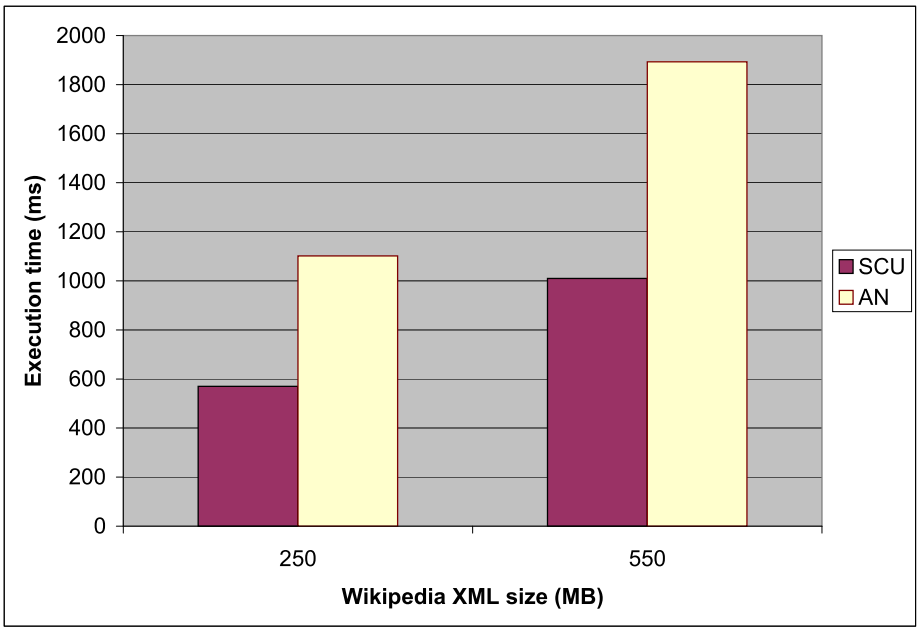


Figure 4.11: Running Q_7 on Wikipedia snapshots of increasing size

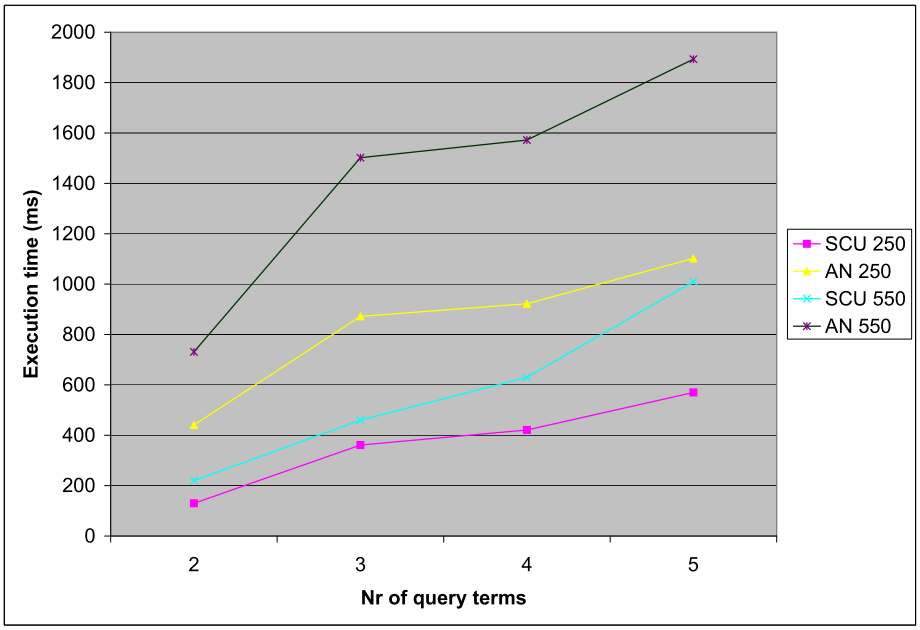


Figure 4.12: Running prefixes of Q_7 on Wikipedia snapshots

4.4.3 In-Depth Evaluation of SCU

In this section, we investigate the effect of query predicates on evaluation time. This issue is highly significant, since matches are materialized only as needed,

namely in selection operators. This strategy delays the computation of the Cartesian product of keyword position lists until a predicate is available, which can then be used to prune matches, effectively turning the Cartesian product into a much cheaper join computation. Our experiments show that this pruning is quite effective.

Effect of Document Size

We again use the family of documents of constant depth and increasing size, by varying the document size by nesting into a chain of depth 10 between 1 and 10 copies of the DBLP document (300MB to 3GB).

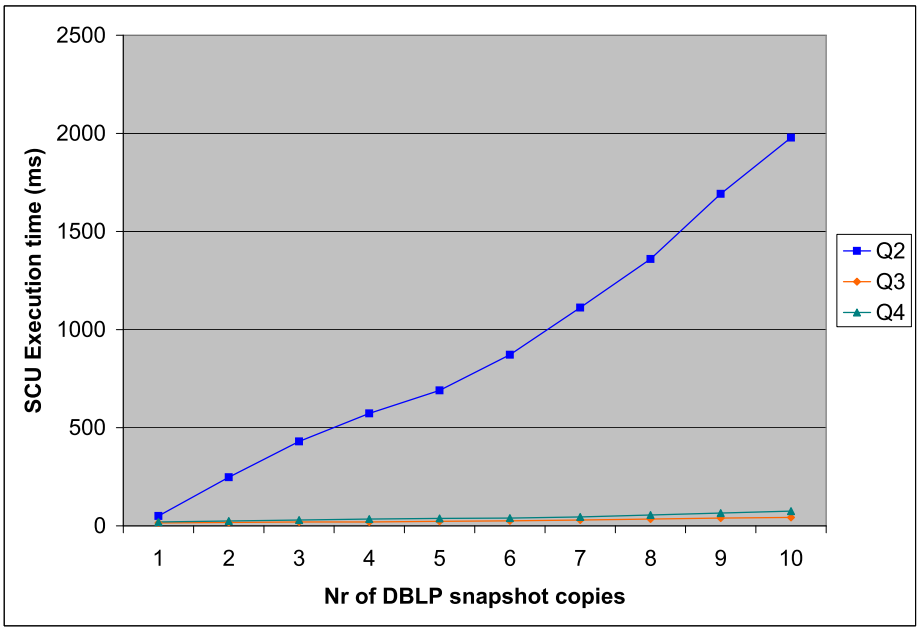


Figure 4.13: Effect of predicate selectivity and document size on match management

Figure 4.13 reports the time to execute the selection predicate only for SCU on the following queries: Q_2, Q_3, Q_4. These queries all share the same conjunctive subquery Q_1, and are distinguished only by the selection predicate. The predicates are chosen to cover the following spectrum of selectivities:

- Q_3's predicate is satisfied by all matches produced by subquery Q_1. This special case is unlikely to occur in practice and is only used as a yardstick giving a lower bound on the evaluation time, since it minimizes the match management work. Indeed, according to the algorithm for selection evaluation, no matches

need be propagated from descendants to ancestors in the SCU table. Moreover, since the first match for each node already satisfies the predicate, there is no need to materialize and test further matches for that node. What is really measured in this case is the stack maintenance overhead.

- Q_2's predicate is violated by all matches returned by Q_1. This is a worst-case scenario which maximizes match management effort. In this scenario, all matches must be propagated to ancestors, and every node requires an exhaustive (and ultimately unsuccessful) search through all its matches. The only reason this search does not result in full materialization of all matches is the aggressive pruning of partial matches as soon as they violate the predicate, saving the work of generating all extensions to full matches for the pruned partial ones. Notice the quasi-linear evolution of the running time with the increase in document size. With a naive implementation without partial match pruning, the running time would have evolved as a polynomial of degree 5 in the number of DBLP snapshots.

- The predicate in Q_4 corresponds to a more typical case: it filters some of the input nodes but not all of them. Notice that in this typical scenario, the running time is quite close to the best case scenario.

Effect of Nesting Depth

This experiment covers the same three representative cases for predicate selectivity as the one in Section 4.4.3. This time, we investigate the effect of nesting depth on the match processing effort. We employ the same queries Q_2, Q_3, Q_4 to this end.

We use the family of documents obtained by nesting a single copy of the DBLP snapshot (300MB) into progressively deeper chains (ranging from depth 1 to 70).

As already explained for the experiment in Section 4.4.2, the processing time does not vary significantly with the document depth, since additional nesting levels do not affect the matches returned by common subquery Q_1, all of which appear solely in the DBLP snapshot. The additional levels do give rise to additional nodes

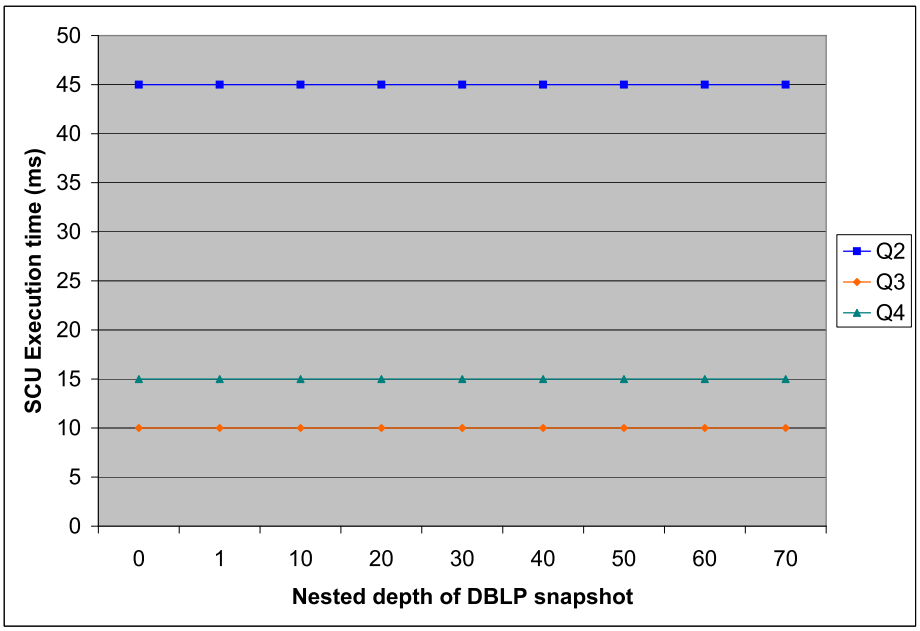


Figure 4.14: Effect of predicate selectivity and nesting depth on match management

in the query answer coming from the chain on top of the DBLP snapshot, but these nodes are not listed in the SCU intermediate result tables since they contain no direct matches.

The running time difference between the 3 queries is due to the selectivity of their predicates: the always true predicate (Q3) generates the least work, the always false predicate (Q2) causes the most, and the sometimes true predicate (Q4) reflects a typical workload.

Exploiting Query Rewritings Opportunities.

This experiments validates one of the main intuitions which drove the design of the XFTalgebra. We targeted a set of operators inspired from relational algebra, such that the beneficial impact of relational query rewritings (e.g. pushing selections into joins) carries over in the full-text algebraic setting.

Figure 4.15 shows the result of running AllNodes and SCU algorithms on two equivalent queries, Q_2 and its optimized counterpart, Q_6 where selection predicates are pushed into the **and** operator. The experiment was run on DBLP documents of varying sizes where the number of snapshots are varied from 1 to

10. It is not surprising to see that performing a relational-like rewriting improves performance for both algorithms.

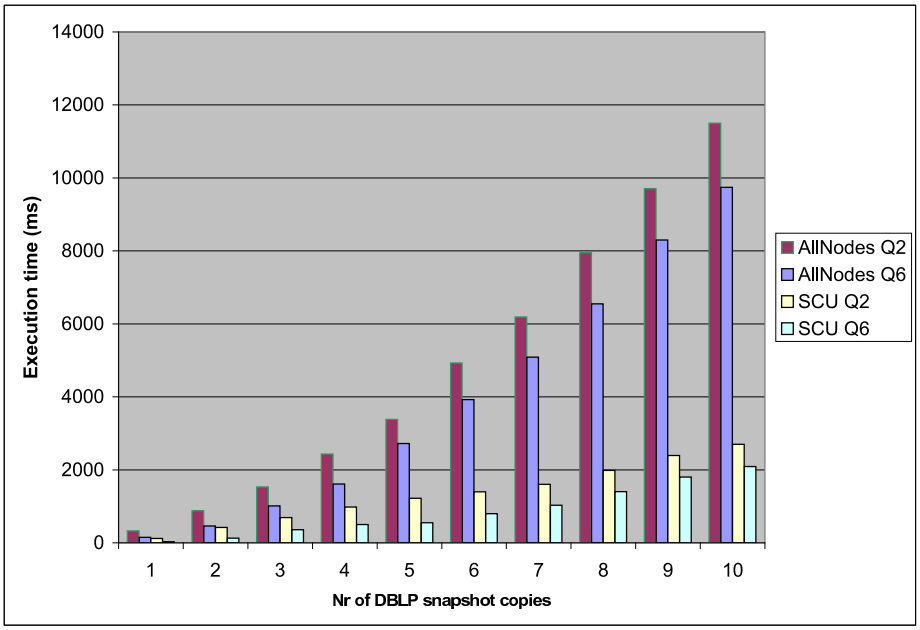


Figure 4.15: Benefit of using relational-like rewritings

4.4.4 Comparison to Existing XML Full-Text Engines

We compared SCU with GalaTexand Quark (Figure 4.16). Due to the limitations of these two systems, we ran simple conjunctive queries (1 to 4 terms) on a 150KB XMark document[6]. GalaTexand Quark have similar performances which are worse than AllNodes and SCU. The performance difference increases with queries containing more terms.

4.5 Related Work

Several full-text algebras and query evaluation algorithms have been proposed in the past [23, 58, 76, 86, 106, 122]. The best-known algebras are text region algebras which were proposed to model structured full-text search [53, 58, 87, 106]. A text region is a sequence of consecutive words in a document and is often used

[6] *http://monetdb.cwi.nl/xml/*

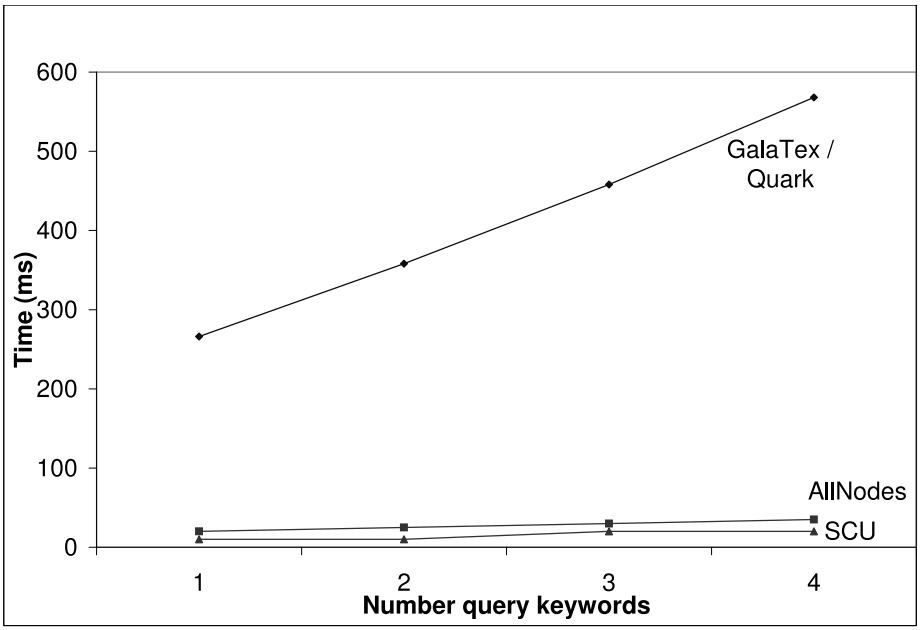


Figure 4.16: AllNodes, SCU, GalaTex and Quark

to represent a structural part such as section and chapter. However, this algebra has limited expressive power [58]. The TIX and TOSS algebras [23, 86] focus on evaluating conjunctive queries and phrase search while the algebra in [122] is used to express NEXI queries [120] and is thus less expressive than XFT.

There has been extensive research in information retrieval on the efficient evaluation of full-text queries [32], including structured full-text queries [45] and of XML queries such as XQuery/IR [44], XSEarch [55], XIRQL [75], XXL [118] and Niagara [128]. However, these works develop algorithms for specific full-text predicates in isolation.

The idea of computing the most specific elements for conjunctive queries has been actively explored using LCAs [80, 92, 109, 125]. We show in Section 4.3 that extending this idea to support the efficient evaluation of queries with complex full-text predicates needs to account for individual term matches in XML elements and, sometimes propagate matches along the XML hierarchy. Moreover, we show that a blind application of state of the art stack-based algorithms [80, 92, 125] results in higher complexities. This is due to the fact that pre-order is a natural choice for XPath evaluation, since any other ordering would require materializing the document in main memory, or a two pass algorithm. However, for full-text, inverted lists are

generated off-line and could be in postorder, which ends up being the natural order expected and automatically preserved by all our algorithms.

Relevance ranking methods for XML are based on extending the well-established vector and probabilistic methods [32] to incorporate structure by propagating answer scores along the XML tree [75], considering overlapping elements [52], applying length normalization to paths [46] or computing tag or path-based term weights [27, 55, 79]. None of them accounts for query predicates to score answers and is thus not applicable to distinguish between the binding and the existential semantics.

4.6 Summary

We presented efficient evaluation algorithms that account for element nesting in XML document structure to evaluate queries with complex full-text predicates. Our algorithms are based on the XFT algebra which subsumes the XQFT-class of full-text languages, enabling a uniform treatment of their optimization and efficient evaluation problems. The novelty of our algorithms lies in their ability to combine relational query evaluation techniques with the stack-based exploitation of element nesting when evaluating full-text predicates. Moreover, they are based on a lazy materialization of term matches which greatly improves query response times. We ran an extensive set of experiments which validate our algorithms. We are currently exploring more efficient match management which involves caching of shared matches in order to further improve the performance of full-text predicates. We believe that the knowledge of the semantics of full-text predicates can be used to devise optimization techniques which are more suited to full-text search. In particular, the order in which individual term matches are explored can have a great impact on proximity operators. This observation directly relates to the problem of devising join algorithms in the relational literature. We believe this work bridges the gap between relational-style optimizations and IR-like predicates and constitutes a good start to the application of well-established relational join techniques to IR query evaluation.

4.7 Acknowledgements

Chapter 4, in part, is a revised reprint of the following material published in SIGMOD 2006 Conference, which is joint work with Sihem Amer-Yahia and Alin Deutsch [25]. This material is also currently being prepared for submission for publication. The dissertation author was the primary investigator and author of the papers.

Chapter 5

Demonstration of XTreeNet

5.1 Introduction

In this Chapter, we describe XTreeNet, a distributed query dissemination engine which facilitates democratization of publishing and efficient data search among members of online communities with powerful full-text queries.

XTreeNet serves as a proof of concept by proposing a novel distributed infrastructure in which data resides only with the publishers owning it, which allows full control over their own content. Expressive user queries are disseminated to publishers. Given the virtual nature of the global data collection, our infrastructure efficiently discovers the publishers that contain matching documents with a specified query, processes the complex full-text query at the publisher, and returns all matching documents to consumer.

While different queries might hit the same set of nodes, our goal is to balance the community search generated load (e.g., number of messages) at nodes during query dissemination while preserving both low space usage of index at a node and minimum information maintained at a node as dictated by the data-location anonymity requirement. Thus, XTreeNet's architecture is based on a novel distributed data index UQDT, organized as a union of overlay (i.e., logical) networks. We show how to use the UQDT infrastructure to achieve overall load balance and maximize the throughput given a workload of rich-expressive queries (i.e., XQuery Full-Text queries) through efficient query routing algorithms and optimization strate-

gies over UQDT. To this end, we leverage the techniques explored in Chapter 2 to disseminate queries efficiently in the community network using the UQDToverlay. We also employ techniques from Chapters 3 and 4 to allow for expressively querying the global community data at the data source.

Moreover, XTreeNet allows users to adjust and tune the system parameters (i.e., the number of overlays, the overlay topology, the query routing strategy) to get insight into the UQDT design trade-offs and into the query evaluation process. Although one of the drivers for our architectural design concerns privacy preservation and data-location anonymity, this is not the main focus of the demo.

Because of the increasing number of online communities and social network sites, the need for powerful search expressivity (beyond simple keyword search) and the need for democratic information exchange by protecting the dissemination infrastructure against censorship, we believe a demonstration of our UQDT infrastructure design, trade-offs, and proposed techniques are of general interest.

The VLDB demo is available at the following location [1]. A preliminary version of the demo [2] is running with limited functionality. This is running a simulator on a single machine that disseminates conjunctive queries, each composed of set of keywords, and returns the set of matching documents.

We exemplify next the UQDT index infrastructure, the distributed query dissemination techniques, and the query evaluation. Then, we present our network system organization that supports efficient construction of multi-overlays (i.e., logical networks). Finally, we detail our XTreeNet demonstration scenarios.

5.2 Processing Full-Text Queries via Distributed Access Methods

Given a user query Q, XTreeNet identifies the relevant data sources that contain matching documents and returns them to the querier. In a first step, the system identifies only the sources of interest and contacts them based on a simplified form of the query Q^S (i.e., the set of keywords in the query). Then, the sources

[1] http://db.ucsd.edu/xtreenet
[2] http://snack.ucsd.edu:8180/xtreenetgui

decide based on their own access policy (i.e., the querier's identity or the history of who accessed what) whether to release the matching documents to the queriers by running the full query Q on their corresponding local document collection. We detail these features next.

5.2.1 Efficient Data Source Discovery

Our indexing solution targets a service-oriented logical network infrastructure, in which we distinguish two types of nodes. There are data *publisher nodes* (the community members) that provide data services and connect to the network via direct links to nodes at its edge. The data are indexed inside the network, which consists of a set of inter-connected and reconfigurable *router nodes*. These are responsible for routing queries to the relevant publishers.

XTreeNet is based on on a novel distributed data index design, called UQDT, that is organized as a union of query dissemination trees (QDT's) realized as an overlay (i.e., logical) network. QDT's purpose is to focus the query forwarding with high probability towards only a subset of all publisher nodes that contain matching documents. Regardless of which querier initiates a query Q, Q is sent to the QDT's root, and it propagates down the tree to the publishers.

For the purpose of information discovery and flexible querying, documents and queries are represented as collections of *content descriptors*, called CDs. A CD is an abstraction of a hierarchical data item. In our case, CDs are keywords in a path context. When the context is empty this corresponds to simple full-text keyword search. The CD collection that describes the user query and the documents are automatically extracted.

We adopt a data partitioning approach in which we partition the global CD collection into multiple partition blocks. Each block is associated with a QDT which takes care of forwarding the data that falls in this partition block. We build smart hierarchical summaries at each node of a QDT to facilitate early pruning by disseminating Q only to the relevant publishers. Each node summary consists of the union of all advertised CDs by publishers in its subtrees for the corresponding QDT. We implement node summaries as counting bloom filters for their well known

properties: compactness and quick probabilistic set membership of CDs.

We employ the following query routing algorithm. Note that the data partitioning scheme determines also a partitioning among the CDs of a query. Therefore the query can be disseminated on all or either of the QDT's that map to the query partitioning blocks. We show in Chapter 2 that it suffices to send the query to the root of only one QDT while still preserving the query semantics. When a router node receives the query message, it forwards it in parallel to each of its children in QDT if and only if the CD set of the query is contained in the node summary. When a publisher node is reached, the full query (not just the CDs extracted from the query) is evaluated on the local database. Any matching documents are sent back to the querier. The advantage of this approach is that the query dissemination is very selective since pruning decision at each node is done on a conjunctive basis (conjunction of CDs).

The number of QDT's to setup in the network and the QDT to send queries are optimization problems for throughput maximization that we solved in Chapter 2. The intuition behind our decision for the number of QDT's is that since the load in one QDT decreases from the root to leaves for any query workload, we try to balance the load of each router node by assigning them to different levels across the different QDT's. One solution which behaves well in practice is to cyclically permute the nodes on tree levels across all QDT's such that all routers appear precisely once in the top levels of any QDT. Picking the QDT to disseminate the query is based on query selectivity estimation techniques. In particular, we avoid routing based on query blocks that contain popular CDs. In Chapter 2 we show that keeping track of a very small number of advertised popular CDs (2-3%), which we call partially informed strategies, XTreeNet achieves almost as good performance as if we had the selectivity information for all the data collection, called the fully informed routing strategy. Note that the latter approach is infeasible in a complete decentralized setting as ours.

5.2.2 Query Evaluation at Source

Without loss of generality, we employ XML data sources. Users publish and query XML repositories. A CD is then the abstraction of a keyword in an XPath context. To be able to express powerful queries over such data we adopt the XQuery Full-Text [114] (XQFT) standard specification which permits composable full-text search primitives such as simple keyword search, Boolean queries, and keyword-distance predicates over XML data. In order to process such expressive class of queries we deploy an XQFT engine processor at each source. In particular, we leverage GalaTex described in Chapter 3 as a processor together with the optimization framework developed in Chapter 4.

The query semantics calls for iteration on all documents in the local repositories at identified relevant sources and for returning only those documents matching with the full XQFT query back to the querier. For efficiency, the backend XQFT engine is build on top of a local index store, which contains posting lists corresponding to documents in the local store at each publisher.

5.2.3 Example

We demonstrate in this section an example of distributed query processing using our infrastructure, the UQDT. First, let us assume a publishing community whose infrastructure connects eight publishers P_1 to P_8, each producing news articles and blogging-aware. Each publisher maintains a local store accessible through an XML interface. To describe their local store and to make their data available for search, the publishers advertise descriptive CDs.

Second, let us remind the query used in the example (‡) from page 3 that asks for:

> *all documents that contain terms related to* `official ethnic` *groups and* `minority` *languages that occur within a window of 10 words, with* `ethnic` *appearing before* `minority` [120, 114].

One way to express this query using the XQuery Full-Text language is as

follows:

Q : *doc()/(blog)*[. **ftcontains** (*"ethnic"* **and** *"minority"* **ordered**) **and** *"official"* **window** ≤ 10 *words*]

Then, the corresponding CDs for Q are the following XPath expressions: $\mathsf{cd}(Q) = \{\ cd_1 = doc/blog/\text{“}ethnic''$, $cd_2 = doc/blog/\text{“}minority''$,
$cd_3 = doc/blog/\text{“}official''\}$

Let us assume now a data partitioning scheme $\mathcal{P}$ over all the published CDs in our community that groups them in at least two distinct partition blocks as follows:

$B_1 = \{doc/blog/\text{“}ethnic'', doc/blog/\text{“}official'', ..\}$,

$B_2 = \{doc/blog/\text{“}minority'', ..\}$, etc.

The partitioning scheme used, $\mathcal{P}$, induces a partitioning $\mathcal{P}_Q$ over the query CDs, $\mathsf{cd}(Q)$, as shown below: $Q_1 = \{doc/blog/\text{“}ethnic'', doc/blog/\text{“}official''\}$ and $Q_2 = \{doc/blog/\text{“}minority''\}$.

Given this partitioning scheme, it automatically implies that the peers in the community network organize themselves into dissemination trees or QDT's – as many as there are partitioning CD blocks, at least two. Let us assume that CD block B_1 corresponds to QDT_1, and B_2 corresponds to QDT_2, respectively. Consequently, $\mathsf{qdt}(Q_1) = QDT_1$ and $\mathsf{qdt}(Q_2) = QDT_2$, where each query block Q_i is associated with a different logical QDT overlay.

Let us also assume that the routing strategy chooses to route the query based on the most selective of the query blocks. To compute the query selectivity for query blocks, we compute the product of individual CD selectivities for the CDs contained in that corresponding block. In this case, if we note with $s(cd)$ as the selectivity of term "cd", then the selectivity of query block Q_1 is $s(Q_1) = s(doc/blog/\text{“}ethnic'') * s(doc/blog/\text{“}official'')$. Consider, in this example, that Q_1 is more selective than Q_2. Therefore, the full query Q is sent for dissemination on the QDT_1 overlay – corresponding to block Q_1. The QDT_1's internal organization as well as the dissemination on QDT_1 based on Q_1's contents is shown in Figure 5.1. We recall that dissemination on the QDT is realized based on checking bloom filter membership on a simplified form Q, which we call Q^S, instead of using

the full Q. In this example, we have that $Q^S = Q_1$.

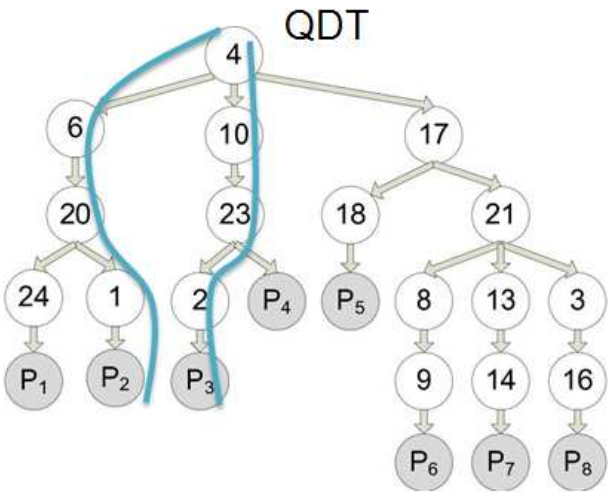


Figure 5.1: Routing Q using Q_1 on QDT_1. Full query Q is evaluated at the candidate relevant publishers P_2 and P_3, respectively.

Each node in the tree contains a summary, implemented as a bloom filter, with all CDs advertised under its subtrees. We considered that CDs in $\mathsf{cd}(Q_1)$ are only advertised by publishers P_2 and P_3 and therefore, they appear in the summaries for the nodes: 4, 6, 10, 20, 23, 1, and 2. Figure 5.1 shows the dissemination path of Q_1 down to the relevant publishers. At each step, QDT_1 is checking bloom filter membership between the CDs in Q_1 with the CDs at each touched node n, or $\mathsf{cd}(Q_1) \subseteq \mathsf{cd}(n)$, by using the node summary to the validate the membership test $n.\mathsf{smm}^M_{QDT_1}.\mathsf{contains}(cd(Q_1))$.

Note that P_2 and P_3 is a superset of the publishers that can actually return matching documents, acting as a candidate set for the relevant publishers. When the query Q_1 reaches a candidate publisher, that node (in this case leaf nodes P_2 and P_3) runs the full XQFT query Q (i.e., all structural and full-text conditions besides the conjunctive CD check realized with the bloom filters during the query dissemination process) locally using GalaTex. Any matching documents to Q are returned back to the querier. This ensures that the XQFT query semantics is preserved, while our methodology is sound and complete. That is our system discovers and returns all documents published in the community matching to complex XML full-text queries according to publisher's advertised specification with no false positives.

5.3 Demonstration Scenario

We propose to demonstrate interactively all the functionality of XTreeNet. The purpose of this demo is a proof-of-concept to show the new architectural design in action as well as the flexibility and efficiency of distributed query processing; in particular, the efficiency in identifying relevant sources and XQFT query processing at the source.

To obtain a true-to-life community, we consider a distributed community that shares a real data collection, namely an XML dump of Wikipedia, comprising about 1.1 million real Wikipedia documents which amount to 8.6 GB [65].

To facilitate the peers interconnectivity in the community we use a 3^{rd} party set of logical routers. We can imagine the alternative solution as well, in which each peer publishes data, queries and forwards queries.

We describe next the underlying network infrastructure and the overlay network support. We then present the demonstrated features.

5.3.1 Network Infrastructure

The XTreeNet distributed engine relies on an overlay network to manage the distributed index across the XTreeNet participants. The index consists of multiple, custom designed QDT's, where each node contains a counting bloom filter. Each filter continuously updates the index by processing the stream of updates from its child nodes. Because a bloom filter is an in-network *aggregate*, it may be computed efficiently across these trees. However, XTreeNet's wide-area deployment requires the maintenance of the bloom filters to be scalable, network-aware and failure resilient. To achieve these operating criteria, the peer-2-peer overlay can employ any off-the-shelf DHT-based technology as backend for overlay functionality (i.e., basic data storage and lookups) and fault tolerance. In practice, XTreeNet uses Mortar [93], an in-house stream processing engine, which is a platform for instrumenting distributed end-hosts with stream operators that facilitates in-network efficient aggregations (e.g., Bloom filter operations) at peer nodes.

Mortar provides a number of features that are important for XTreeNet. First, it supports user-defined in-network aggregates, allowing XTreeNet to create

hierarchical index summaries by simply extending Mortar with custom stream operators that implement the spectral bloom filter. Each operator indexes local content and propagates index changes to its parent. Mortar also arranges the nodes in the tree in a network-aware fashion, ensuring that the majority of participants are within a low-latency horizon of the root. Further, unlike DHT-based in-network aggregation systems, Mortar allows applications to control the design of additional trees. This is critical, as XTreeNet balances the query load by controlling the shape of the set of trees implementing the index. Other features provide fast and reliable operator management (installation/removal) and accurate stream processing in the presence of node failures and unsynchronized clocks.

We evaluate XTreeNet by deploying 1,000 to 10,000 XTreeNet peers over an emulated network using the ModelNet [121] emulator. Modelnet is a large-scale evaluation environment that combines the realism of an Internet testbed with the ability to execute experiments in a reproducible fashion. In Modelnet, unmodified applications run over unmodified operating systems and network stacks, while the emulator subjects the application's traffic to the bandwidth, delay and loss constraints of the emulated network topology. In our setting, 34 physical machines, running Linux 2.6.9 and connected on a Gigabit network, emulate an Internet-like topology built with the Inet [8] topology generator.

5.3.2 Running Scenario

The XTreeNet query interface at a peer is very simple, yet it hides a powerful and efficient architectural design. The web browser interface accepts keyword search queries in the XQuery Full-Text style allowing for a multitude range of queries supporting from simple keyword queries, keywords in a context (XPath) to complex queries with predicates on the keyword positions (e.g., proximity distance predicates).

In addition to the query input, the interface allows a user to tune the various system parameters: setting the number of QDT's in the UQDT structure, the QDT to route a query on, or the routing state amount maintained at each node to guide the routing process.

The result of executing a query is a page containing links to the matching

documents together with a detailed section containing statistical and routing information decisions. Such information includes the suggested routing QDT, both the processing and the forwarding loads generated in the system as the number of total exchanged messages and a break down on execution times for the various stages of query processing. We enumerate in the following various demo scenarios of interest.

Query Routing and Processing in XTreeNet. First, we show the interactive functionality to search the global community data collection by executing different ad-hoc XQFT queries. We demonstrate the use of UQDT as a distributed index infrastructure that provides support for complex querying via multiple index lookups during query forwarding and then running an XQFT processor at the publisher.

After a peer issues a query, we present a visualization of the internal routing flow of the query into the network. First, the system decomposes the query into query blocks based on the UQDT partitioning scheme over the keywords. By using these query blocks, we show how the system smartly decides the corresponding QDT overlay where the query is actually routed on. We show next the routing path of the query on that QDT by specifying the nodes and the links touched. At the end of routing, the reaching leaves are the candidate publishers. Let us notice that this is a super set of relevant publishers since they have been selected based on the conjunctive part of the query.

Finally, the full query is tested at these publishers by running an XQFT processor to check and retrieve the actual matching documents. The querier has the option of interactively changing the set of matching documents, simply by varying the keyword conditions in the query (e.g., adding or removing keywords and keyword predicates).

Democratization of Publishing. The ability for an individual publisher to dynamically control the access to the content she owns, including the ability to make visible what information she wants to, on a selective basis to different users, is a highly desirable aspect in a democratic network.

A key aspect of such an information access infrastructure, that is a key to the democratization of the Internet, is to make all requests for information (queries) available to all the publishers who may have relevant information and allowing them

to determine their response to the query.

We show how publishers can maintain control over their own data. Depending on what they decide to advertise, independent on the actual local published content, different queries may reach them. In this demo we do not enforce a particular publisher policy to access their data even though one can think of sophisticated ways of doing it. We consider, for now, that publishers answer all received queries correctly.

Moreover, we do not consider for the scope of this chapter the system's robustness to failures, as the main focus is to show how to leverage the new UQDT index structure for efficient distributed query processing and privacy in P2P publishing. We defer this as future work. However, preliminary analysis shows that this is feasible based on the Mortar infrastructure (as described in Section 5.3.1).

Our DLA-preserving UQDT infrastructure prevents leaking any information about which publishers are capable of answering a given query. In the case of compromised nodes, we allow to zoom into individual routers and introspect the actual information that is being kept as part of the UQDT index. We show that getting hands on the local bloom filter and the overlay connections at a router does not reveal much information.

Interactive Tuning. As we mentioned previously, we let the user interact with the main parameters of the system in order to get a feeling of the various trade-offs.

For instance, the querier can choose to disseminate queries over specific QDT overlays. When a peer issues a query, the system suggests what would be the best query routing strategy based on techniques described in Section 5.2.1. However, the querier can send the query to any of the existing QDT overlays for the following reasons: she may have domain or external knowledge on the selectivity of the overall published data items, or may want to analyze generated traffic on different QDT dissemination, or just has a preference for a particular set of nodes.

In the case that the querier decides not to send the query to the suggested QDT, we run the query on both variants (on the suggested QDT as well as on the querier-chosen QDT) and even though the number of answers is the same, we show a comparison of the amount of generated traffic.

Similarly, the querier may change the amount of routing state. This can lead to changes in the suggested QDT among the available QDT overlays. Intuitively, the more state is maintained at a node, the more precise the suggested QDT is relative to the best routing (e.g., when all the selectivity state of published data items is known).

Balancing the load. In this scenario, we show that XTreeNet can be used to balance the overall traffic. The load is near-optimum uniformly distributed among the peers.

We run a query workload based on a uniform distribution of queries with the number of conjuncts varying from 1 to 10. We stress the load such that each conjunctive query has a match in the global data collection. We let the user dynamically setup the UQDT configuration by picking the number of QDT's. For choosing the overlay topology we employ off-the-shelf tools developed by network research for multicasting (e.g., generated by SCRIBE [47]).

During the query workload execution, we collect statistical information. At the end, we report processing and forwarding load histograms between different number of QDT's. We show that for 15-QDT's the load is well balanced in the system and therefore, the overall throughput is maximized. We define the throughput as the number of queries answered per unit of time. At the same time, in the 1-QDTcase this shows the high congestion in the system.

5.4 Discussion: System Architecture

The advantage of our 2-part solution is in-line with the current Database technology to de-couple the different layers of representation and execution. For instance, the beauty of XTreeNet is the ability to dissociate the backend network from the frontend system.

First, we provide a distributed backend infrastructure that plays the role of an index to speed up access to data, which is similar to what we see in traditional Database systems. To this end, we propose the UQDT data structure that consists of a distributed data index. The index stores and provides quick lookup functionality for terms, which are generically called CDs, and their cojunctions at the lowest

level of granularity of a data collection. In the context of semi-structured document collections, a CD is represented as a keyword in a structural context (i.e., XPath path context).

Second, the frontend consists of an XQFT query processor, which resides locally at each data source. The XQFT query processor execution relies on it own local index for more specialized fast access, being dependant on the frontend query language. In our case, we employed and support XQuery Full-Text (XQFT) queries. Therefore, the frontend's execution is independent from backend's functionality. The advantage of this modular, yet integrated, approach is the ability of the lower level to support different query language abstractions (e.g., SQL, XQuery, XQFT, etc.) with little or no modifications.

Moreover, whereas the current search engine technology or the existing hosted online community are still a viable solution for Internet search, they are not entirely suited for democratic community-based publishing due to their increasing requirements, mainly performance, user privacy-related issues, and true ability to freely exchange data without the fear of discrimination and user censorship. Yet, our XTreeNet platform does not target to replace the mainstream technologies entirely. On the contrary, we believe our system can complement the functionality of centralized publishing and search platforms at times where controversial or sensitive publishing situation calls for it; or where the user feels the need for more privacy in the sense of anonymity without worrying about their identity being associated with the published content.

We envision a practical hybrid platform where both type of systems can coexist. A centralized solution would benefit by the general-purpose, mainstream existing search applications where privacy risk to participating users does not represent a potential harm to anyone. We propose a complementary approach to search, in the form of a decentralized search platform, UQDT, that encourages users and enables true online free flow of information as well as full user autonomy while keeping control over their own published data. Therefore, we militate for this model as a much better way to provide freedom of speech online among autonomous members; thereby, protecting users from censorship, harassment, and possible discrimination by third parties.

5.5 Conclusion

In this chapter we presented XTreeNet, an efficient infrastructure that empowers information publishers to join censorship-resistant communities and query their global data collection in an ad-hoc fashion using expressive queries.

5.6 Acknowledgements

Chapter 5, in part, is a revised reprint of the material published in VLDB 2008 Conference, which is joint work with Alin Deutsch, Dionysios Logothetis, K.K. Ramakrishnan, Divesh Srivastava, and Kenneth Yocum [64]. The material was accompanied by a real implementation and a system demonstration during the Demonstration track at VLDB 2008 Conference.

Chapter 6

Conclusion and Future Work

6.1 Concluding Remarks

This thesis advances the state-of-the-art in understanding how to bridge the gap between XML Information Systems and Databases at the intersection with Distributed Information Systems. In this thesis, we provide an integrated solution to democratic community-based publishing and searching the community data using flexible, composable, and expressive full-text queries.

As new XML applications continue to emerge on the Web, there is a need for a higher degree of sophistication and flexibility in the way data is produced, managed, and queried. Such examples range from digital libraries to enterprise level content management, scientific applications, and search in online communities. Given this rapid growth in user application needs in online communities to freely exchange information and to search ad-hoc for content, as well as the increasing awareness for user privacy while participating in the publishing process, we address critical issues relevant to nowadays online publishing systems.

Mainly, we propose a novel efficient publishing infrastructure platform that empowers publishers to join democratic online communities and empowers users to query the global data collection in an ad-hoc fashion using expressive queries over semi-structured data. To this end, we addressed and solved problems related to how publishers join the community infrastructure and make their data accessible for search, how to freely exchange data among community users, how publishers

can answer complex queries locally efficiently with consistent scores. At the same time, we also answer consumer related problems such as how to ask semi-structured data expressively and whom to direct user queries in order to find the publishers, and thereof matching documents, of interest based on ad-hoc user-specified full-text conditions.

Our solution consists in designing and implementing a decentralized system, called XTreeNet, which grants community users to keep control over their own data and which supports queries against the global data collection, with no need for a central authority that disintermediates publishers from consumers. In particular, the focus of the dissertation consists in the following two aspects.

First, we leverage our distributed UQDT index infrastructure as a platform to disseminate queries in the community from consumers to relevant publishers. Since the dissemination indices are subject to potential external interceptions and attacks, our approach precludes third parties from learning the exact associations between publishers and advertised CD data without compromising a significant portion of the community network. Our contributions range from identifying the design space with its trade-off dimensions, relevant metrics and notion of optimality, to introducing solutions that achieve near-optimality with only low overhead.

Our UQDT index is based on judiciously designed load balanced dissemination overlay trees and on carefully chosen query routing strategies that use query selectivity estimation techniques. We showed that partially-informed routing is promising as a best routing strategy with low overhead cost yielding similar results as the ideal, fully-informed routing strategy. The solution exploits crucially the dual role of QDT's as both query dissemination and as statistics tree overlays. While we showed that fanout-balanced tree overlays are closest to optimal, an advantage of our solution is its generality, in the sense that it focuses on extracting the performance inherent in any given topology.

Second, while UQDT index supports CD-level (i.e., keywords or keywords in context) index lookups only, we empower XTreeNet to support complex filtering conditions on the CD matches that go beyond simple keyword search to be applied at the publisher site. One representative query language of this class of queries is W3C's XQuery Full-Text (XQFT) standard specification. This provides functionality for

composable predicates such as Boolean keyword search and different flavor keyword-based distance conditions (i.e., distance, window, order) as well as the number of times to repeat a condition. XQFT provides support for full integration of structured search with the full-text conditions over semi-structured data, or XML.

Our contribution lies in designing and implementation of a universal query optimization framework for the XQFT-class of languages that is deployed as part of an XQFT evaluation engine at publishers to resolve full-text queries over semi-structured data collections efficiently. To this end, we presented efficient evaluation algorithms that account for element nesting in XML document structure.

In particular, our optimization framework is based on an algebraic formalism of full-text languages in terms of patterns of keywords and a relational way to manipulate the matches of these patterns in the document collection. The advantage of our approach is that it enables a relational-style evaluation as well as the true-and-tried relational-style optimizations for well-behaved scoring functions. As such, our algorithms are based on the XFT algebra which subsumes the XQFT-class of full-text languages, enabling a uniform treatment of their evaluation and optimization problems. The novelty of our algorithms lies in their ability to combine relational query evaluation techniques with stack-based exploitation of element nesting when evaluating full-text predicates. We believe this work bridges the gap between relational-style optimizations and information retrieval (IR) like predicates and constitutes a good start to the application of well-established relational optimization techniques to IR query evaluation.

Ultimately, to show the viability of our approach, we have built XTreeNet, a real distributed search infrastructure that seamlessly integrates the two components: query routing and query evaluation.

6.2 Future Work

This dissertation takes an important step forward towards understanding design principles and performance issues by bridging the gap, in a principled manner, between information retrieval and database information systems with a focus on distributed information systems.

One of the key challenges that we pointed out in this thesis is the lack of solutions for the integration of structured and unstructured search over semi-structured data collections. We believe our solution is a significant step in this direction as we explored XQFT processing techniques at the publisher nodes. This includes a generic evaluation and optimization framework for serving complex XML queries with composable full-text predicates. Similarly, we have also investigated a generic approach of implementing XQFT on top of an XQuery engine via user defined scoring functions. In order to expose more and to share functionality across the two languages, it would be valuable to open up the black-box XQFT user defined functions to the XQuery implementation. This will permit for better coupling the corresponding languages at the algebraic level. Therefore, we would like to better understand the full integration (i.e., tighter than via user defined functions) of XQFT processing with the various XQuery evaluation engines in order to explore new opportunities for cross-language optimization.

Another big challenge we discussed in this thesis is the users concern for privacy in online communities in order to fully deliver on free speech and free information exchange without the worry of being the target of censorship, discrimination, or harassment. We have shown in XTreeNet a complementary way to achieve such guarantees. We believe it is challenging to combine our solution with existing methods including encrypted communication channels and anonymizer servers to provide an end-to-end privacy resistance against third parties.

Another promising research direction is to incorporate ranking functions and distributed scoring to exploit the UQDT overlay for top-k query processing (i.e., retrieving only the k top ranked results for a given query). The very nature of decentralized control in P2P makes it difficult to employ directly today's search engines ranking methods, where extensive global analysis and personalized methods are used. On the good side, P2P IR systems lend themselves to ranking various other methods based on reputation, access frequencies, peer authority, etc. This implies that it would also be interesting to devise a generic full-text evaluation and optimization platform that could accommodate not just one way to score query answers, but various such existing application-specific relevance ranking functions.

Consequently, the growth, the sophistication and complexity, and the hetero-

geneity of available data call for more advanced techniques to manage information, and especially to search and query data. In general, we argue that a tighter integration of databases and distributed information retrieval systems would provide better performance, effectiveness, and functionality by combining the best of the worlds.

For instance, the techniques developed in this thesis are geared more towards querying homogeneous semi-structured data sources using structured query language paradigms with composable and complex querying primitives over distributed online communities. This applies to data collections of known and similar schemata. However, real-life data consists of combining data from various data sources with different schemata. Understanding the role of heterogeneity in managing multi- non-integrated individual data sources on the Web, is key to providing uniform search over various types of data (e.g., structured, semi-structured, and unstructured). Thus, it is interesting to explore novel and alternative search paradigms to provide not just flexible and expressive search, but also semantic search support as well as guide the user formulate complex data analytics queries that can handle data heterogeneity. We believe that key here is to establish formal grounds and formal methods for such querying to enable progress and development of theory and practical paradigms and their optimization solutions. To this end, we have made the first steps in this direction as in works like [33, 34, 35], where we adopt a "pay-as-you-go" approach, as promoted by "dataspaces" [81], to assist searching, exploring, discovering, and analysing complex non-integrated heterogeneous data sources.

Index

References

[1] BIRN: Biomedical Informatics Research Network. http://www.nbirn.net/.

[2] Blogger. http://www.blogger.com/.

[3] DBLP in XML. http://dblp.uni-trier.de/xml/.

[4] Facebook. http://www.facebook.com/.

[5] Friendster. http://www.friendster.com/.

[6] GEON: National Geosciences Cyberinfrastructure Network. http://www.geongrid.org/.

[7] Health Level Seven. http://www.hl7.org/.

[8] Inet. http://topology.eecs.umich.edu/inet.

[9] INitiative for the Evaluation of XML Retrieval. http://inex.is.informatik.uni-duisburg.de/.

[10] Jap: The jap anonymity & privacy homepage (2000-2005). http://www.anon-online.de/.

[11] Library of Congress. http://xml.house.gov/.

[12] LinkedIn. http://www.linkedin.com/.

[13] LiveJournal. http://livejournal.com/.

[14] Myspace. http://www.myspace.com/.

[15] National Coalition Against Censorship (NCAC). http://ncac.org//.

[16] OpenNet Initiative (ONI). http://opennet.net//.

[17] Pastry: A Substrate for Peer-to-peer Applications. http://freepastry.org/.

[18] Text REtrieval Conference. http://trec.nist.gov/.

[19] Twitter. http://www.twitter.com/.

[20] Wikipedia. http://www.wikipedia.org/.

[21] WordPress. http://www.wordpress.com/.

[22] S. Abiteboul, I. Dar, R. Pop, G. Vasile, D. Vodislav, and N. Preda. Large Scale P2P Distribution of Open-source Software. In *International Conference on Very Large Data Base (VLDB Demo)*, 2007.

[23] S. Al-Khalifa, C. Yu, and H. V. Jagadish. Querying Structured Text in an XML Database. In *ACM International Conference on Management of Data (SIGMOD)*, 2003.

[24] S. Amer-Yahia, C. Botev, and J. Shanmugasundaram. TeXQuery: A Full-Text Search Extension to XQuery. In *International World Wide Web Conference (WWW)*, 2004.

[25] S. Amer-Yahia, E. Curtmola, and A. Deutsch. Flexible and Efficient XML Search with Complex Full-Text Predicates. In *ACM International Conference on Management of Data (SIGMOD)*, 2006.

[26] S. Amer-Yahia, M. Fernandez, D. Srivastava, and Y. Xu. Phrase Matching in XML. In *International Conference on Very Large Data Base (VLDB)*, 2003.

[27] S. Amer-Yahia, N. Koudas, A. Marian, D. Srivastava, and D. Toman. Content and Structure Scoring for XML. In *International Conference on Very Large Data Base (VLDB)*, 2005.

[28] S. Amer-Yahia, L. Lakshmanan, and S. Pandit. FleXPath: Flexible Structure and Full-Text Querying for XML. In *ACM International Conference on Management of Data (SIGMOD)*, 2004.

[29] R. Anderson. The Eternity Service. In *Pragocrypt*, 1996.

[30] M. Artigas, P. Lopez, J. Ahullo, , and A. Skarmeta. Cyclone: A Novel Design Schema for Hierarchical DHTs. In *IEEE International Conference on Peer-to-Peer Computing*, 2005.

[31] B. Babcock and C. Olston. Distributed Top-K Monitoring. In *ACM International Conference on Management of Data (SIGMOD)*, 2003.

[32] R. Baeza-Yates and B. Ribiero-Neto. Modern Information Retrieval. In *Addison-Wesley*, 1999.

[33] A. Balmin, L. Colby, E. Curtmola, Q. Li, and F. Ozcan. Search Driven Analysis of Heterogenous XML Data. In *Conference on Innovative Data Systems Research (CIDR)*, 2009.

[34] A. Balmin, L. Colby, E. Curtmola, Q. Li, F. Ozcan, S. Srinivas, and Z. Vagena. SEDA: A System for Search, Exploration, Discovery and Analysis of XML Data. In *International Conference on Very Large Data Base (VLDB Demo)*, 2008.

[35] A. Balmin and E. Curtmola. WikiAnalytics: Ad-hoc Querying of Highly Heterogeneous Structured Data. In *International Conference on Data Engineering (ICDE Demo)*, 2010.

[36] A. Balmin, V. Hristidis, N. Koudas, Y. Papakonstantinou, D. Srivastava, and T. Wang. A System for Keyword Proximity Search on XML Databases. In *International Conference on Very Large Data Base (VLDB)*, 2003.

[37] G. Banavar, T. Chandra, B. Mukherjee, N. Nagarajarao, R. Strom, , and D. Sturman. An Efficient Multicast Protocol for Content-based Publish-Subscribe Systems. In *International Conference on Distributed Computing Systems (ICDCS)*, 1999.

[38] M. Bender, S. Michel, P. Triantafillou, G. Weikum, , and C. Zimmer. Minerva: Collaborative P2P Search. In *International Conference on Very Large Data Base (VLDB)*, 2005.

[39] B. Bloom. Space/time Trade-offs in Hash Coding with Allowable Errors. *Communications of the ACM (CACM)*, 13(7), July 1970.

[40] A. Bonifati, U. Matrangolo, A. Cuzzocrea, and M. Jain. XPath Lookup Queries in P2P Networks. In *ACM International Workshop on Web Information and Data Management (WIDM)*, 2004.

[41] C. Botev, S. Amer-Yahia, and S. Shanmugasundaram. Expressiveness and Performance of Full-Text Search Languages. In *International Conference on Extending Database Technology (EDBT)*, 2006.

[42] P. Boucher, A. Shostack, and I. Goldberg. Freedom Systems 2.0 Architecture. In *White paper, Zero Knowledge Systems, Inc.*, 2000.

[43] A. Bozdog, R. van Renesse, and D. Dumitriu. Selectcast: A Scalable and Self-repairing Multicast Overlay Routing Facility. In *ACM Workshop on Survivable and Self-regenerative Systems*, 2003.

[44] J. M. Bremer and M. Gertz. XQuery/IR: Integrating XML Document and Data Retrieval. In *International Workshop on the Web and Databases (SIGMOD WebDB)*, 2002.

[45] E. W. Brown. Fast Evaluation of Structured Queries for Information Retrieval. In *International ACM Conference on Reasearch and Development in Information Retrieval (SIGIR)*, 1995.

[46] D. Carmel, Y. S. Maarek, M. Mandelbrod, Y. Mass, and A. Soffer. Searching XML Documents via XML Fragments. In *International ACM Conference on Reasearch and Development in Information Retrieval (SIGIR)*, 2003.

[47] M. Castro, P. Druschel, A.-M. Kermarrec, , and A. Rowstron. Scribe: A Large-scale and Decentralized Application-level Multicast Infrastructure. In *IEEE Journal on Selected Areas in Communications (JSAC)*, 2002.

[48] R. Chand and P. Felber. A Scalable Protocol for Content-based Routing in Overlay Networks. In *Proc. of Symposium on Network Computing and Applications*, 2003.

[49] S. Chaudhuri. An Overview of Query Optimization in Relational Systems. In *ACM Symposium on Principles of Database Systems (PODS)*, 1998.

[50] M. Cherniack and S. Zdonik. Rule Languages and Internal Algebras for Rule-based Optimizers. In *ACM International Conference on Management of Data (SIGMOD)*, 1996.

[51] T. T. Chinenyanga and N. Kushmerick. Expressive and Efficient Ranked Querying of XML Data. In *International Workshop on the Web and Databases (SIGMOD WebDB)*, 2001.

[52] C. Clarke. Controlling Overlap in Content-Oriented XML Retrieval. In *International ACM Conference on Reasearch and Development in Information Retrieval (SIGIR)*, 2005.

[53] C. Clarke, G. Cormack, and F. Burkowski. An Algebra for Structured Text Search and a Framework for its Implementation. *The Computer Journal (CJ)*, 38(1):43–56, 1995.

[54] I. Clarke, O. Sandberg, B. Wiley, and T. Hong. A Distributed Anonymous Information Storage and Retrieval System. In *Lecture Notes in Computer Science (LNCS)*, 2001.

[55] S. Cohen, J. Mamou, Y. Kanza, and Y. Sagiv. XSEarch: A Semantic Search Engine for XML. In *International Conference on Very Large Data Base (VLDB)*, 2003.

[56] W. Cohen. Integration of Heterogeneous Databases Without Common Domains Using Queries Based on Textual Similarity. In *ACM International Conference on Management of Data (SIGMOD)*, 1998.

[57] G. Conforti, G. Ghelli, P. Manghi, and C. Sartiani. Scalable Query Dissemination in XPeer. 2007.

[58] M. P. Consens and T. Milo. Algebras for Querying Text Regions: Expressive Power and Optimization. *Journal of Computer and System Sciences (JCSS)*, 57(3):272–288, 1998.

[59] A. Crainiceanu, P. Linga, A. Machanavajjhala, J. Gehrke, and J. Shanmugasundaram. P-Ring: An Efficient and Robust P2P Range Index Structure. In *ACM International Conference on Management of Data (SIGMOD)*, 2007.

[60] A. Crespo and H. Garcia-Molina. Routing Indices for Peer-to-peer Systems. In *International Conference on Distributed Computing Systems (ICDCS)*, 2002.

[61] E. Curtmola, S. Amer-Yahia, P. Brown, and M. F. Fernández. GalaTex: A Conformant Implementation of the XQuery Full-Text Language. In *International Workshop on XQuery Implementation, Experience and Perspectives (SIGMOD XIME-P)*, 2005.

[62] E. Curtmola, S. Amer-Yahia, P. Brown, and M. F. Fernández. GalaTex: A Conformant Implementation of the XQuery Full-Text Language. In *International World Wide Web Conference (WWW Special interest tracks and posters)*, 2005.

[63] E. Curtmola, S. Amer-Yahia, and A. Deutsch. Implementation and Open Research Issues in XML Full-Text Search. In *New York Area DB/IR Day 2005*, April 2005.

[64] E. Curtmola, A. Deutsch, D. Logothetis, K. Ramakrishnan, D. Srivastava, and K. Yocum. XTreeNet: Democratic Community Search. In *International Conference on Very Large Data Base (VLDB Demo)*, 2008.

[65] L. Denoyer and P. Gallinari. The Wikipedia in XML Corpus. In *International ACM Conference on Reasearch and Development in Information Retrieval (SIGIR)*, 2006.

[66] Y. Diao, S. Rizvi, and M. Franklin. Towards an Internet-scale XML Dissemination Service. In *International Conference on Very Large Data Base (VLDB)*, 2004.

[67] R. Dingledine, M. Freedman, and D. Molnar. The Free Haven Project: Distributed Anonymous Storage Service. In *International Workshop on Designing Privacy Enhancing Technologies*, 2001.

[68] R. Dingledine, N. Mathewson, and P. Syverson. TOR: The Second-generation Onion Router. In *USENIX Annual Technical Conference*, 2004.

[69] D. Eastlake and P. Jones. SHA1: US Secure Hash Algorithm 1. In *IETF - Network Working Group Report RFC3174*, 2001.

[70] L. Fan, P. Cao, J. Almeida, and A. Broder. Summary Cache: A Scalable Wide-Area Web Cache sharing Protocol. *IEEE/ACM Transactions on Networking*, 8(3), 2000.

[71] W. Fenner, M. Rabinovich, K. K. Ramakrishnan, D. Srivastava, and Y. Zhang. XTreeNet: Scalable Overlay Networks for XML Content Dissemination and Querying. In *International Workshop on Web Content Caching and Distribution (WCW)*, 2005.

[72] M. Fernández, J. Siméon, B. Choi, A. Marian, and G. Sur. Implementing XQuery 1.0: The Galax Experience. In *International Conference on Very Large Data Base (VLDB)*, 2003.

[73] D. Florescu, D. Kossmann, and I. Manolescu. Integrating Keyword Search into XML Query Processing. In *International World Wide Web Conference (WWW)*, 2000.

[74] M. Freedman and R. Morris. Tarzan: A Peer-to-peer Anonymizing Network Layer. In *ACM Conference on Computer and Communications Security (CCS)*, 2002.

[75] N. Fuhr and K. Grossjohann. XIRQL: An Extension of XQL for Information Retrieval. In *International ACM Conference on Reasearch and Development in Information Retrieval (SIGIR)*, 2000.

[76] N. Fuhr and T. Rölleke. A Probabilistic Relational Algebra for the Integration of Information Retrieval and Database Systems. *ACM Transactions on Information Systems (TOIS)*, 15(1), 1997.

[77] L. Galanis, Y. Wang, S. Jeffery, and D. DeWitt. Locating Data Sources in Large Distributed Systems. In *International Conference on Very Large Data Base (VLDB)*, 2003.

[78] P. Ganesan, K. Gummadi, and H. Garcia-Molina. Canon in G Major: Designing DHTs with Hierarchical Structure. In *International Conference on Distributed Computing Systems (ICDCS)*, 2004.

[79] T. Grabs and H. Schek. ETH Zürich at INEX: Flexible Information Retrieval from XML with PowerDB-XML. In *INitiative for the Evaluation of XML Retrieval Workshop (INEX)*, 2002.

[80] L. Guo, F. Shao, C. Botev, and J. Shanmugasundaram. XRANK: Ranked Keyword Search over XML Documents. In *ACM International Conference on Management of Data (SIGMOD)*, 2003.

[81] L. Haas, D. Kossmann, E. Wimmers, and J. Yang. Optimizing Queries across Diverse Data Sources. In *International Conference on Very Large Data Base (VLDB)*, 1997.

[82] M. Harren, J. Hellerstein, R. Huebsch, B. Loo, S. Shenker, and I. Stoica. Complex Queries in DHT-based Peer-to-Peer Networks. In *International Workshop on Peer-To-Peer Systems (IPTPS)*, 2002.

[83] Y. Hayashi, J. Tomita, and G. Kikui. Searching Text-rich XML Documents with Relevance Ranking. In *SIGIR Workshop on XML and Information Retrieval*, 2000.

[84] J. M. Hellerstein, J. F. Naughton, , and A. Pfeffer. Generalized search trees for database systems. In *International Conference on Very Large Data Base (VLDB)*, 1995.

[85] V. Hristidis, Y. Papakonstantinou, N. Koudas, and D. Srivastava. Keyword Proximity Search in XML Trees. *IEEE Transactions on Knowledge and Data Engineering (TKDE)*, 18(4), 2006.

[86] E. Hung, Y. Deng, and V. S. Subrahmanian. TOSS: An Extension of TAX with Ontologies and Similarity Queries. In *ACM International Conference on Management of Data (SIGMOD)*, 2004.

[87] J. Jaakkola and P. Kilpelainen. Nested Text-Region Algebra. In *Report C-1999-2, Dept. of Computer Science, University of Helsinki*, 1999.

[88] H. Jagadish, B. Ooi, K. Tan, Q. Vu, and R. Zhang. BATON*: Speeding Up Search in Peer-to-peer Networks with a Multi-way Tree Structure. In *ACM International Conference on Management of Data (SIGMOD)*, 2006.

[89] A. Kirsch and M. Mitzenmacher. Less Hashing, Same Performance: Building a Better Bloom Filter. In *Lecture Notes in Computer Science (LNCS)*, volume 4168 of *European Symposium on Algorithms (ESA)*, 2006.

[90] G. Kokkinidis and V. Christophides. Semantic Query Routing and Processing in P2P Database Systems. In *International EDBT Workshop on Peer-to-Peer Computing and Databases (P2P&DB)*, 2004.

[91] G. Koloniari and E. Pitoura. Content-based Routing of Path Queries in Peer-to-peer Systems. In *International Conference on Extending Database Technology (EDBT)*, 2004.

[92] Y. Li, C. Yu, and H. V. Jagadish. Schema-Free XQuery. In *International Conference on Very Large Data Base (VLDB)*, 2004.

[93] D. Logothetis and K. Yocum. Wide-Scale Data Stream Management. In *USENIX Annual Technical Conference*, 2008.

[94] D. Lomet. Replicated Indexes for Distributed Data. In *International Conference on Parallel and Distributed Information Systems (PDIS)*, 1996.

[95] A. Manjhi, V. Shkapenyuk, K. Dhamdhere, and C. Olston. Finding (Recently) Frequent Items in Distributed Data Streams. In *International Conference on Data Engineering (ICDE)*, 2005.

[96] P. M.F. An Algorithm for Suffix Stripping. 1980.

[97] A. Mislove and P. Druschel. Providing Administrative Control and Autonomy in Peer-to-peer Overlays. In *International Workshop on Peer-To-Peer Systems (IPTPS)*, 2004.

[98] S.-H. Myaeng, D.-H. Jang, M.-S. Kim, and Z.-C. Zhoo. A Flexible Model for Retrieval of SGML Documents. In *International ACM Conference on Reasearch and Development in Information Retrieval (SIGIR)*, 1998.

[99] M. Ott, L. French, R. Mago, and D. Makwana. XML-based Semantic Multicast Routing: An Overlay Network Architecture for Future Information Services. In *IEEE GLOBAL COMMUNICATIONS CONFERENCE (GLOBECOM)*, 2004.

[100] O. Papaemmanouil and U. Cetintemel. SemCast: Semantic Multicast for Content-based Data Dissemination. In *International Conference on Data Engineering (ICDE)*, 2005.

[101] H. Piresh, J. Hellerstein, and W. Hasan. Extensible/rule based Query Rewrite Optimization in Starburst. In *ACM International Conference on Management of Data (SIGMOD)*, 1992.

[102] R. Ramakrishnan and J. Gehrke. Database Management Systems. In *McGraw-Hill, 3rd Ed.*, August 2002.

[103] M. Rennhard and B. Plattner. Practical Anonymity for the Masses with Morphmix. In *Lecture Notes in Computer Science (LNCS)*, Financial Cryptography, 2004.

[104] S. Rhea and J. Kubiatowicz. Probabilistic Location and Routing. In *IEEE Conference on Computer Communications (INFOCOM)*, 2002.

[105] S. Robertson. The Probability Ranking Principle in IR. In *Journal of Documentation*, volume 33, 1977.

[106] A. Salminen and F. Tompa. PAT Expressions: an Algebra for Text Search. In *Acta Linguistica Hungar. 41 (1-4)*, 1992.

[107] G. Salton, , and A. Wong. A Vector Space Model for Automatic Indexing. In *Communications of the ACM (CACM)*, volume 18, 1975.

[108] G. Salton and M. J. McGill. Introduction to Modern Information Retrieval. In *McGraw-Hill*, 1983.

[109] A. Schmidt, M. Kersten, and M. Windhouwer. Querying XML Documents Made Easy: Nearest Concept Queries. In *International Conference on Data Engineering (ICDE)*, 2001.

[110] I. Tatarinov, S. Viglas, K. Beyer, J. Shanmugasundaram, E. Shekita, and C. Zhang. Storing and Querying Ordered XML using a Relational Database System. In *ACM International Conference on Management of Data (SIGMOD)*, 2002.

[111] The World Wide Web Consortium. Extensible Markup Language (XML) 1.0 (Fifth Edition). http://www.w3.org/TR/xml/.

[112] The World Wide Web Consortium. XML Path Language (XPath) 2.0. W3C Working Draft. http://www.w3.org/TR/xpath20/.

[113] The World Wide Web Consortium. XQuery 1.0: An XML Query Language. http://www.w3.org/TR/xquery/.

[114] The World Wide Web Consortium. XQuery 1.0 and XPath 2.0 Full-Text. Working draft. http://www.w3.org/TR/xquery-full-text/.

[115] The World Wide Web Consortium. XQuery 1.0 and XPath 2.0 Functions and Operators. W3C Working Draft. http://www.w3.org/TR/xquery-operators/.

[116] The World Wide Web Consortium. XQuery and XPath Full-Text Requirements. W3C Working Draft. http://www.w3.org/TR/xmlquery-full-text-requirements/.

[117] The World Wide Web Consortium. XQuery and XPath Full-Text Use Cases. W3C Working Draft. http://www.w3.org/TR/xmlquery-full-text-use-cases/.

[118] A. Theobald and G. Weikum. The Index-Based XXL Search Engine for Querying XML Data with Relevance Ranking. In *International Conference on Extending Database Technology (EDBT)*, 2002.

[119] P. Triantafillou and T. Pitoura. Towards a Unifying Framework for Complex Query Processing over Structured Peer-to-Peer Data Networks. In *Databases, Information Systems and Peer-to-Peer Computing (DBISP2P)*, 2003.

[120] A. Trotman and B. Sigurbjörnsson. NEXI, Now and Next. In *INitiative for the Evaluation of XML Retrieval Workshop (INEX)*, 2004.

[121] A. Vahdat, K. Yocum, K. Walsh, P. Mahadevan, D. Kostic, J. Chase, and D. Becker. Scalability and Accuracy in a Large-scale Network Emulator. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2002.

[122] J. Vittaut, B. Piwowarski, and P. Gallinari. An Algebra for Structured Queries in Bayesian Networks . In *INitiative for the Evaluation of XML Retrieval Workshop (INEX)*, 2004.

[123] M. Waldman and D. Mazieres. Tangler: A censorship-resistant publishing system based on document entanglements. In *ACM Conference on Computer and Communications Security (CCS)*, 2001.

[124] M. Waldman, A. Rubin, and L. Cranor. Publius: A Robust, Tamper-evident, Censorship-resistant and Source-anonymous Web Publishing System. In *USENIX Annual Technical Conference*, 2000.

[125] Y. Xu and Y. Papakonstantinou. Efficient Keyword Search for Smallest LCAs in XML Databases. In *ACM International Conference on Management of Data (SIGMOD)*, 2005.

[126] Z. .Xu, R. Min, and Y. Hu. HIERAS: A DHT Based Hierarchical P2P Routing Algorithm. In *International Conference on Parallel Processing (ICPP)*, 2003.

[127] P. Yalagandula and M. Dahlin. A Scalable Distributed Information Management System. In *ACM SIGCOMM Conference*, 2004.

[128] C. Zhang, J. Naughton, D. DeWitt, Q. Luo, and G. Lohman. On Supporting Containment Queries in Relational Database. In *ACM International Conference on Management of Data (SIGMOD)*, 2001.

[129] S. Zoels, M. Eichhorn, A. Tarlano, and W. Kellerer. Content-based Hierarchies in DHT-based Peer-to-Peer Systems. In *International Symposium on Applications on Internet Workshops (SAINT-W)*, 2006.